Escuela Politécnica Superior
Departamento de Informática

Proyecto de Fin de Carrera

bsync: Sistema extensible para la sincronización y copia de seguridad de ficheros para su uso en organizaciones

Javier Torres Niño

Tutores
Juan Miguel Gómez Berbís
Alejandro Rodríguez González

28 de Julio de 2011

Resumen

En la actualidad existen muchos sistemas de sincronización y copia de seguridad de ficheros, pero la mayoría de ellos no comercializan el producto, optando por comercializar los servicios ofrecidos por medio de estos. Esto impide el uso de estas herramientas a entidades que necesiten que sus datos no salgan de su entorno, por lo que el objetivo de este Proyecto de Fin de Carrera es el diseño y desarrollo de un sistema similar a estos otros productos.

Se analizan las características de los competidores, y se diseña un programa que incluye en el núcleo las características básicas y sea capaz de incluir todas las características adicionales mediante el uso de extensiones o plugins. Es precisamente esta extensibilidad la que caracteriza el sistema, permitiendo que un mismo producto pueda adaptarse a las necesidades de multitud de clientes, que pueden precisar, por ejemplo, de distintos sistemas de autenticación (para integrar el sistema en la arquitectura existente) o utilizar la infraestructura para sincronizar más que ficheros.

Palabras clave: sincronización, copia de seguridad, extensibilidad, plugins, sistemas distribuidos

Agradecimientos

Es impensable dejar este proyecto sin unas breves líneas en las que agradecer a todas las personas que me han soportado durante estos últimos cinco años, y especialmente los últimos meses.

En primer lugar tengo que mencionar a todos los compañeros de clases con los que tanta risas he compartido, especialmente a todos los integrantes de vastralis, que tan unidos hemos estado estos últimos años. Sin vosotros no habría logrado llegar hasta aquí.

También quiero recordar a mis amigos de toda la vida, ya sabéis quiénes sois. ¿Que sería de mí sin esas largas noches sentados todos alrededor de una mesa, jugando y de cháchara?

No puedo olvidar mencionar a todo el equipo docente que ha hecho posible que pueda completar este proyecto, sin vuestras enseñanzas no tendría ni la mitad de conocimientos que he utilizado durante su realización. Llegué a la universidad creyendo que sabía bastante de informática, ahora me voy con la certeza de no saber casi nada.

También se merecen una mención especial mis compañeros de laboratorio, cotutores oficiales y extraoficiales de este proyecto. Sin vuestros consejos y lo que me aprendí de vosotros sobre el campo investigador, este proyecto habría sido muy distinto.

Y en último lugar, aunque no por ello menos importante, a mi familia por introducirme en el mundo de la informática a la tierna edad de los 6 años, a la que me he agarrado desde entonces, primero como hobby y después como profesión. Mención especial para mi hermano por aguantar mi escaqueo generalizado de las tareas del hogar mientras terminaba de escribir esta memoria.

A todos, muchas gracias
Javier

Índice general

Índice de figuras

Índice de tablas

Índice de listados de código

Parte I

Introducción

Capítulo 1

Introducción

1.1. Motivación

Tras la popularización de los computadores la información se ha convertido en el recurso más valioso, no hay empresa o gobierno que pudiera sobrevivir en su forma actual sin los sistemas informáticos para tratar y almacenar esta información. Sin embargo, en muchas ocasiones la protección de esos datos se releva a un segundo plano con lo que los accidentes producen pérdidas con alto impacto económico.

La solución principal para proteger estos datos lleva siendo la misma desde los albores de la informática: la copia de seguridad. Es un mecanismo muy sencillo a nivel técnico pero un gran problema a nivel logístico por la dificultad de identificar correctamente los datos que deben ser asegurados, la frecuencia de las copias, el sistema de almacenamiento para estas y muchas otras consideraciones.

Es por estas dificultades que, por lo general, los planes de copias de seguridad sólo incluyen los datos almacenados en servidores y otros repositorios de información centralizados por almacenarse en estos los datos más críticos, ignorando los datos almacenados en las estaciones de trabajo de los usuarios: informes, contratos y otros documentos que a menudo son tan importantes para el funcionamiento diario como las bases de datos principales.

El problema también afecta a los usuarios particulares, que en su gran mayoría no realizan copias de seguridad, algunas veces por no disponer de la capacidad de almacenamiento para ello y otras porque es difícil recordar realizar la copia de seguridad manualmente, y los sistemas de copia de seguridad tradicionales a menudo consumen demasiados recursos como para considerar su ejecución automática.

Por otra parte, en los últimos años, el gran desarrollo en redes de comunicación y en concreto de Internet, permite formas totalmente nuevas de colaboración remota. La disponibilidad en tiempo real de la información de un grupo de trabajo permite acelerar muchos procesos, evitando los antiguos retrasos producidos por la necesidad de enviar por correo o fax los documentos tras cada modificación. Muchas soluciones existen para facilitar estos procesos, pero en su forma más básica la mayoría consisten en un sistema de ficheros en red para compartir la información entre las distintas estaciones de trabajo, aunque a menudo también se añaden otras funcionalidades para facilitar la compartición

de contactos o documentos.

La combinación de ambas ideas, la copia de seguridad de datos y la sincronización en tiempo real en un sólo programa es una idea reciente, popularizada con servicios que aprovechan las capacidades de la nube para almacenar la inmensa cantidad de datos manejada por estos servicios, que no sólo almacenan la copia de trabajo de los ficheros, si no también todas las anteriores, actuando como un sistema de copia de seguridad instantáneo.

En ámbitos que necesiten mayor privacidad y no deseen que sus datos pasen por manos de terceros, se pueden aprovechar los servidores utilizados por herramientas colaborativas para realizar copias de seguridad de los datos de las estaciones de trabajo, almacenando tanto las copias de trabajo como las antiguas en este servidor.

Con este tipo de sistemas se combinan la ventajas de ambos sistemas y se facilita la difícil tarea de la copia de seguridad de los datos de las estaciones de trabajo. Sin embargo, la mayoría, si no todas, las alternativas existentes utilizan el modelo de servicios como su modelo de negocios, por lo que el funcionamiento del sistema se ve ligado a la utilización de su plataforma de almacenamiento de datos. Aunque los precios suelen ser muy competitivos, esto no soluciona el problema de la privacidad de los datos, puesto que estos son almacenados por terceros.

Algunos servicios solucionan parte de este problema cifrando los datos antes de enviarlos al servidor de almacenamiento, utilizando una contraseña sólo conocida en el cliente lo que asegura que el proveedor de servicio no tiene acceso a los datos. Esto no soluciona el problema de la dependencia de la disponibilidad de un servicio ajeno. Además, al ser plataformas muy cerradas, todas comparten el mismo defecto principal, la inflexibilidad.

Ninguno de los sistemas existentes de este tipo permite modificar o extender las capacidades del producto para adaptarlo a necesidades propias del usuario. Por ejemplo: un usuario podría querer utilizar uno de estos servicios para sincronizar sus contactos, lo que podría conseguirse extendiendo el programa cliente para que tomara contactos de varias fuentes y los sincronizara como un fichero más puesto que, al fin y al cabo, no son más que datos. Otros ejemplos serían la necesidad de una empresa de integrar el producto con sus sistemas internos o la adición de un sistema de chat para facilitar la colaboración entre un equipo de trabajo modificando los mismos datos.

Son estas deficiencias las que motivan la creación de un nuevo sistema que comparte funcionalidad con muchos otros, pero que pretende facilitar la extensión de sus capacidades, creando una plataforma más abierta que las existentes.

1.2. Objetivos

El objetivo de este Proyecto de Fin de Carrera es el diseño y desarrollo de uno de estos sistemas híbridos de colaboración y copia de seguridad, de ahora en adelante, de sincronización de ficheros. El objetivo principal del diseño es un sistema muy extensible, de forma que pueda utilizarse en multitud de situaciones distintas y pueda adaptarse a necesidades futuras sin necesidad de modificar el núcleo del programa. Para ello, se propone un sistema de extensiones o plugins, que permitan añadir funcionalidad al programa principal.

Para ello, se evaluarán los sistemas existentes y sus características, clasificándolas en esenciales y opcionales, de forma que las esenciales pararán a formar parte del núcleo del programa, mientras que se proveerán las capacidades necesarias para poder desarrollar las características opcionales como extensiones del programa.

Además de la mencionada extensibilidad, es necesario tener en cuenta otros factores de cara al diseño, destacando la seguridad de los datos (disponibilidad, confidencialidad y control de acceso) o el rendimiento del sistema (para permitir una sincronización de los datos lo más cercana posible al tiempo real).

1.3. Metodología

Al tratarse de un proyecto de una sola persona, las metodologías tradicionales para el desarrollo de proyectos informáticos no son adecuadas, puesto que están pensadas para un equipo de mayores dimensiones, que es el motivo de la estructura tan rígida y el amplio uso de documentos como forma de intercambiar ideas que las caracteriza.

Por ello, una metodología ágil es más adecuada para este proyecto, aunque teniendo siempre en cuenta que las reuniones entre miembros del equipo son innecesarias, aunque siguen siendo útil la comunicación con el jefe de equipo, cuya figura está representada por los tutores del proyecto en este caso.

Muchas de estas técnicas utilizan prototipos extensivamente, pero sin embargo esto no es muy adecuado a este proyecto porque es difícil crear protocolos con poca funcionalidad para ir ampliándolo. Esto se debe a que, al tratarse de un sistema complejo, cada iteración del prototipo debe incluir el mismo conjunto de características tanto en el desarrollo del software como en el protocolo de red. Además, la funcionalidad mínima apreciable es la sincronización de ficheros, incluyendo la notificación de cambios, transferencias de ficheros, etc. Esto supone una gran parte del sistema, con lo que el prototipo inicial se proporcionaría en un momento muy tardío del proyecto. Esto no significa sin embargo que no puedan utilizarse prototipos, sólo que una metodología basada en ellos no es la más adecuada.

Se decide utilizar una metodología que utiliza muchos principios de las metodologías ágiles como son las reuniones frecuentes con el tutor o el desarrollo incremental mezclando las fases de diseño e implementación, tomando también principios de metodologías clásicas como es la del desarrollo en espiral, cuyo esquema se muestra en la figura 1.1.

Para ello, en primer lugar se realizará un análisis del estado del arte y de los principales problemas a decidir en el proyecto, especialmente aquellos que afecten a todo el sistema como son la arquitectura y el protocolo de comunicaciones a utilizar.

Una vez se dispone de este análisis, se divide el trabajo en tareas, componiéndose cada una de ellas de cuatro partes: análisis del problema y estado del arte relacionado, diseño, implementación y pruebas, incluyendo si es posible la creación de un prototipo, especialmente en fases más tardías del proyecto como ya se ha comentado. Se intentará que estos pasos sean incrementales, de manera que no sea necesaria o una fase de integración al final o se minimice su complejidad, incorporando la mayor parte de la integración dentro de las tareas correspondientes.

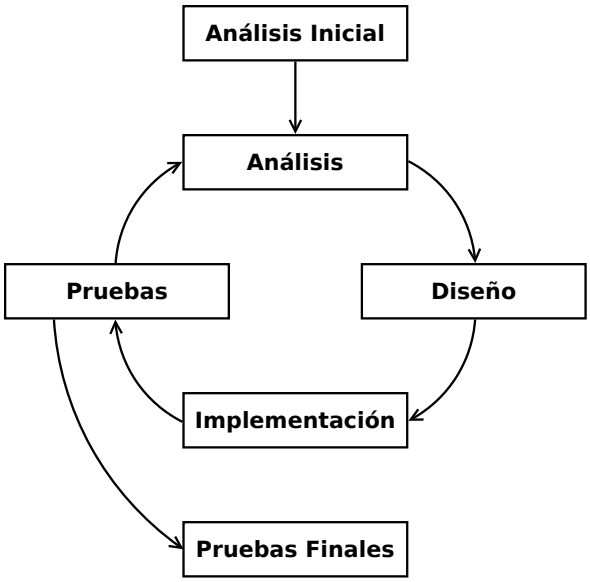


Figura 1.1: Esquema del ciclo de desarrollo utilizado

Finalmente, es posible que sea necesaria una fase de integración de ciertos componentes del sistema que no hayan sido integrados anteriormente, así como las pruebas finales del sistema como conjunto con las correspondientes correcciones en caso de que estas fueran necesarias.

1.4. Organización de este documento

En la parte II se desarrolla el análisis inicial del problema. El capítulo 2 se presenta el estado del arte sobre la sincronización y copia de seguridad de ficheros, incluyendo técnicas de copia de seguridad, sistemas de ficheros distribuidos, algoritmos de sincronización y productos similares al que trata este proyecto, es decir, el estado del arte del sistema como un todo.

El capítulo 3 trata las decisiones más importantes del sistema, incluyendo la elección de la plataforma de desarrollo o la arquitectura de red del sistema, añadiendo más estado del arte de las distintas posibilidades en el caso de ser necesario.

La parte III trata del diseño del prototipo, incluyendo diagramas de clases y modelos de la base de datos de todos los sistemas desarrollados. La información se divide en capítulos correspondientes a los módulos del sistema, no a las tareas en las que se divide el desarrollo del sistema.

El capítulo 4 contiene los detalles de diseño del protocolo de comunicaciones utilizado por el sistema, incluyendo las necesidades descritas en la sección correspondiente del capítulo de análisis.

Los capítulos 5 y 6 describen en detalle el diseño del cliente y del servidor respectivamente, incluyendo tanto el diseño de módulos y clases que componen cada uno de ellos como las bases de datos necesarias en ambos casos.

El capítulo 7 trata de los distintos plugins que pueden utilizarse como el sistema, incluyendo las interfaces que estos deben implementar y el diseño de

las implementaciones de ejemplo en los casos en los que se han incluido.

La interfaz web diseñada para permitir el acceso al sistema sin utilizar el cliente de sincronización se describe en el capítulo 8, junto a las correspondientes decisiones de diseño.

También se incluyen un análisis separado de las principales decisiones de rendimiento y seguridad, así como los *tradeoffs* entre estas dos características en el capítulo 9, de forma que las principales decisiones tomadas a este respecto queden aglutinadas en un único capítulo en vez de dispersas por los múltiples capítulos de la sección de diseño, dado que muchas de ellas afectan tanto al cliente como al servidor.

Finalmente, la parte IV contiene las conclusiones del proyecto y personales, así como la bibliografía y referencias citadas a lo largo de todo el documento en el capítulo 10 y el presupuesto del proyecto en el capítulo 11.

Si está leyendo este documento en su versión electrónica, puede hacer click en los nombres de las referencias a capítulos, secciones, figuras, referencias, etc. para ir directamente a estas. Puede probarlo en las anteriores referencias a capítulos y partes.

Parte II

Análisis del problema

Capítulo 2

Estado del arte

2.1. Copias de seguridad

La copia de seguridad tiene el objetivo de mantener copias de todos los datos de un sistema, de forma que en caso de fallo de almacenamiento en el sistema principal, se pueda recuperar rápidamente minimizando el tiempo de avería. En [1] se da un repaso general a las técnicas de copias de seguridad, clasificándolas según varios criterios, entre los que destacan:

Total o incremental

Mientras que las copias totales incluyen la totalidad de los datos, permitiendo una recuperación sin información adicional, las incrementales incluyen sólo los cambios respecto a la copia anterior. De esta forma se minimiza el espacio ocupado por la copia de seguridad, pero a cambio es necesario disponer de todas las copias incrementales desde la última copia total para poder recuperar los datos.

Otra opción es la combinación de estas técnicas, realizar copias incrementales en periodos cortos de tiempo, mientras que se ejecutan copias de seguridad totales en periodos mayores, por ejemplo: copias incrementales diarias y totales cada semana. Esto combina ventajas de las dos técnicas, reduciendo el espacio ocupado (y normalmente también el tiempo [2]) por las copias sin necesidad de remontarse mucho tiempo para poder recuperar una copia.

De fichero o de dispositivo

Las copias de seguridad de ficheros consisten en copiar cada fichero individualmente al medio de copia de seguridad, de forma equivalente a una operación de copia tradicional. Esto permite recuperar cada fichero por separado de forma sencilla. Sin embargo, debido al funcionamiento de los sistemas de ficheros, cada fichero está distribuido en varios bloques del disco, no necesariamente adyacentes, por lo que una operación de lectura de un fichero suele ser más lenta que una lectura secuencial de los datos del disco.

Así aparecen las copias de seguridad a nivel de dispositivo, que consisten en copiar los datos del dispositivo directamente, incluyendo todas las estructuras

del sistema de ficheros. Esta operación se puede realizar con lecturas secuenciales, con lo que el rendimiento suele ser mayor, pero complica la restauración de ficheros sueltos.

Copias de seguridad en línea

Las copias de seguridad pueden realizarse bloqueando el acceso al sistema de ficheros original. Esto es adecuado en caso de que pueda permitirse la indisponibilidad del sistema durante el proceso, ya que asegura que no habrá problemas de consistencia durante la copia.

En otro caso, se pueden realizar las copias en línea, mientras el sistema puede acceder a los datos, pero se introduce el problema de la modificación de los datos durante la copia. No sólo afecta esto a modificaciones de ficheros durante la copia de éste (en cuyo caso se podría producir un fichero corrupto) si no que es necesario un sistema de escanear el sistema origen durante la copia para detectar ficheros nuevos o borrados, si se desea que la copia refleje estos cambios de última hora. La complejidad del sistema dependerá del grado de consistencia deseado, puesto que mientras que algunos sistemas pueden funcionar con consistencia a nivel de fichero (los ficheros copiados son correctos), otros pueden necesitar que se mantenga consistencia entre todos los ficheros en un mismo directorio (porque estén relacionados), etc.

Otros

Otros temas interesantes tratan de la compresión de copias de seguridad, que además de reducir el espacio pueden mejorar el rendimiento en caso de que las operaciones de lectura/escritura sean más lentas que las de compresión; el medio físico de almacenamiento a utilizar, copias *off-site* a través de la red, etc.

2.2. Sincronización de ficheros

Un problema clásico en los grupos de trabajo es la necesidad de compartir documentos de manera que se pueda trabajar cómodamente con ellos y se mantengan sincronizados de la manera más automática posible. Para esto existen varias entre las que destacan los sistemas de ficheros en red, algoritmos de espejo (*mirroring*) de datos y sistemas de sincronización con copia local.

2.2.1. Sistemas de ficheros en red

Los sistemas de ficheros en red proporcionan la capacidad de acceder a ficheros situados en otros equipos comunicados por alguna red de comunicaciones, de forma que todos los equipos pueden realizar cambios sobre los mismos datos. El sistema de ficheros en red más extendido en el mundo Unix es NFS [3], mientras que en plataformas Windows prevalece SMB (también conocido como CIFS) [4]. Existen muchos más sistemas pero todos se basan en las mismos procesos básicos que incluyen apertura de un *share* y un sistema de llamadas que permita realizar todas las tareas asociadas a los sistemas de ficheros: creación y borrado de ficheros directorios, lectura y escritura de datos y metadatos, listado de directorios, etc.

El problema más difícil de solucionar en estos sistemas es el de la consistencia en las operaciones, sobre todo bajo modificación simultánea de un fichero por varios equipos. Existen varias soluciones para esto, similares a las usadas en sistemas de ficheros locales para controlar el acceso por varios procesos, como por ejemplo los bloqueos de un fichero que permiten al dueño de dicho bloqueo asegurarse la escritura o el acceso en exclusiva.

También es importante el tema del rendimiento, cuya solución suele pasar por usar una caché local de ficheros, ya que las operaciones a través de la red son mucho más costosas que en un sistema local. Sin embargo, el uso excesivo de caché puede causas problemas de consistencia por mantener los datos en caché después de que hayan sido modificados por otros procesos.

Y por último, nunca hay que olvidar la seguridad, de forma que los servicios de seguridad necesarios se cumplan, incluyendo la confidencialidad y el control de acceso, normalmente solventado con cifrado y credenciales o la disponibilidad de los datos, para lo que conviene que el servidor no realice un trabajo excesivo, puesto que esto podría utilizarse para bloquear el servicio con pocas peticiones.

2.2.2. Sincronización periódica o *mirroring* de directorios

Otra aproximación a la sincronización de datos, especialmente si los datos cambian poco o sólo en un equipo es la sincronización periódica de los datos, es decir, la copia a través de la red de dichos datos, de manera que todos los equipos dispongan de la misma versión de ellos. Esto no se diferencia en exceso de las copias de seguridad tradicionales, excepto por el propósito que se persigue y la dificultad de copiar grandes cantidades de datos a través de una red con ancho de banda limitado.

Es por esto que se utilizan sistemas similares a la copia de seguridad incremental, pero que evitan tener que tener la versión anterior del fichero para poder detectar los cambios. El algoritmo más conocidos para esto es rsync [5], que utiliza hashes de los datos del fichero para detectar los cambios. El funcionamiento básico consiste en tomar un bloque de tamaño arbitrario de datos del fichero y enviar su hash, de forma que se pueda determinar si el bloque ha cambiado o no. Además, mediante el uso de *rolling hashes* se consigue que en caso de hacer una inserción al fichero no sea necesario retransmitir todo el fichero (puesto que todos los bloques tras la inserción cambiarían), si no que este cambio puede detectarse y resolverse con la simple transmisión de los datos insertados.

2.2.3. Sincronización con copia local

Esta categoría incluye aquellos sistemas que permiten la sincronización de datos con un servidor central, pero que mantienen una copia local de los datos, utilizable en caso de fallo del servidor, combinando las ventajas de los sistemas de ficheros en red (sincronización inmediata y automática) con la de los sistemas sincronización de directorios (disponibilidad local de los datos).

Estos sistemas se han popularizado recientemente, con la explosión de las nubes de computación que proporcionan almacenamiento de tamaño dinámico a bajo coste, con lo que se han creado multitud de empresas que proporcionan esta clase de software unido al servicio de almacenamiento de datos. Por otro lado, se han creado herramientas open source que ofrecen el producto como tal, siendo el usuario el responsable de proporcionar el servicio.

De entre estas herramientas destacan las siguientes por popularidad o características ofrecidas:

Dropbox Dropbox [6] ofrece el que es probablemente el servicio más popular de esta categoría ya que, a pesar de no ofrecer características avanzadas, ofrece varios gigabytes de almacenamiento de forma gratuita a sus usuarios, además de un servicio de de pago para los usuarios con necesidad de garantías de disponibilidad o mayor espacio.

Ubuntu One Ubuntu One [7] ofrece el código del cliente como software libre, que permite la modificación del código para adaptarse a las necesidades del cliente. Sin embargo, al no ofrecerse el software del servidor, no se puede construir una plataforma privada completa a partir de este servicio.

Tarsnap Tarsnap [8] es uno de los pocos servicios que ofrece cifrado de los datos en el cliente, de forma que el proveedor del servicio no puede analizar los datos subidos. Además, al ofrecer el código del cliente, se puede comprobar su funcionamiento, para garantizar que los datos son cifrados. Sin embargo, la sincronización debe ejecutarse manualmente o con un demonio de notificaciones como lsyncd [9].

Jungle Disk Jungle Disk [10] ofrece la alternativa comercial más completa, incluyendo cifrado y herramientas de colaboración para equipos de trabajo. Sin embargo, no ofrece ningún servicio de manera gratuita y todo su código es privado.

SparkleShare SparkleShare [11] es la alternativa completamente libre más desarrollada en el momento, ofreciendo las mismas capacidades básicas que sus competidores directos, pero ninguna adicional. Utiliza cualquier servidor de git como almacenamiento, con lo que el usuario puede usarlo como solución privada, o utilizar alguno de la multitud de servicios que ofrecen hospedaje de git.

En la tabla 2.1 se muestran las principales diferencias entre las herramientas analizadas.

Producto	**Cliente**	**Servidor**	**Backend**	**Sincro.**[1]	**Cifrado**
Dropbox	Prop.[2]	Prop.	Amazon S3	Sí	No
Ubuntu One	OS[3]	Prop.	Amazon S3	Sí	No
Tarsnap	OS	Prop.	Amazon S3	No	Sí
Jungle Disk	Prop.	Prop.	Amazon S3	Sí	Sí
SparkleShare	OS	OS	git + IRC	Sí	No

Tabla 2.1: Comparación de productos para la sincronización de ficheros

[1] Sincronización, detección automática de cambios
[2] Propietario
[3] Open Source

Capítulo 3

Análisis

El objetivo de este proyecto es desarrollar una solución similar a las comentadas en la sección de sincronización local del estado del arte pero orientada al uso privado y corporativo, es decir, proporcionando un producto en lugar de un servicio y con capacidades de extensibilidad y configuración para adaptarlo a las necesidades personales de cada usuario. En este capítulo se discuten los aspectos a tener en cuenta a la hora de diseñar un sistema con las características deseadas.

3.1. Objetivos principales

En primer lugar, deben tenerse en cuenta los objetivos principales del sistema en orden de importancia, puesto que muchas decisiones dependerán de estos, ya que normalmente cada una de las opciones entre las que elegir tiene ventajas en ciertos campos e inconvenientes en otros.

El público objetivo de este proyecto son las organizaciones, ya que se pretende desarrollar un producto, en vez de un servicio. Esto es más adecuado para empresas que disponen de sus propios sistemas de información donde instalar los programas necesarios para el funcionamiento del sistema, a la vez que es menos adecuado para usuarios finales que prefieren no tener que alojar servidores.

Dado que cada organización tiene requisitos muy diferentes, el objetivo principal del proyecto es proporcionar la máxima flexibilidad al usuario. Esto incluye añadir opciones de configuración para todos los parámetros susceptibles de ser modificados, así como permitir la extensión de las capacidades del programa.

Para facilitar su adopción en entornos heterogéneos, también es necesarios que el sistema sea completamente multiplataforma, incluyendo al menos las principales plataformas de escritorio: Windows, Mac OS X y Linux. Se considerará una ventaja adicional pero no imprescindible la posibilidad de uso en otras plataformas de escritorio (como otros derivados de Unix: Solaris o la familia BSD) y en plataformas móviles, aunque en estos casos interesa su utilización como servidor o como cliente respectivamente.

El sistema tratará de minimizar la interacción con el usuario, ya que se considera que un sistema de sincronización que actúa de forma correcta debe ser transparente para el usuario, no diferenciándose la utilización de datos sincronizados de aquellos que se encuentran únicamente almacenados de forma local.

Por supuesto, es imposible evitar esto por completo debido a la gran cantidad de problemas que pueden surgir con la utilización de la red. En estos caso, los avisos al usuario deben ser poco intrusivos, y el sistema debe ser capaz de recuperarse automáticamente en caso de error, sincronizando los datos que se hayan modificado mientras el sistema estaba fuera de servicio.

Finalmente, nunca hay que olvidar la seguridad y el rendimiento en sistemas de este tipo. El sistema debe ser suficientemente seguro como para garantizar la confidencialidad de los datos en todo momento. Otros servicios de seguridad que deben proporcionarse son la autenticación y control de acceso a los datos, garantías de integridad de los datos tanto en almacenamiento como en transmisión e intentar maximizar la disponibilidad del sistema, minimizando el tiempo que este se encuentra fuera de servicio.

Por el lado del rendimiento se deben intentar minimizar los recursos utilizados por el cliente y el servidor, con especial énfasis en el cliente. Para ello cabe utilizar técnicas de paralelización para las tareas más pesadas, pero prestando especial atención a no saturar los recursos del sistema, especialmente en el cliente, en el que los datos deben sincronizarse sin afectar al resto de tareas del usuario.

3.2. Extensibilidad

La característica principal del producto es la capacidad de extensión de sus funcionalidades. Es necesario permitir extensibilidad sin necesidad de modificar el código fuente del programa, debido a que esto es generalmente más complicado, propenso a errores e impediría comercializar el producto como software privativo. Es decir, es necesario añadir un sistema que permita la ejecución de código externo al producto.

Existen dos alternativas principales para permitir este tipo de extensibilidad. Una de ellas es la utilización de un lenguaje de scripting que pueda acceder a las características del sistema que sea necesario extender. Tiene la ventaja de que puede controlarse el uso que se hace del sistema por parte de las extensiones, pero la desventaja de tener que crear conexiones del sistema de scripting al módulo principal por cada atributo o método que pueda ser modificado por scripts.

La otra alternativa es permitir la creación de extensiones que se añadirán al sistema mediante carga dinámica de librerías (`.dll` en Windows, `.so` en muchos Unix). Al tratarse de librerías desarrolladas en un lenguaje de programación arbitrario, estas pueden realizar cualquier tarea externa al sistema sin problema alguno. Para la modificación del sistema, este proporciona interfaces que definen las funciones que deben ser implementadas por el plugin y a las que el plugin puede llamar. Esto da máxima flexibilidad a cambio de menor seguridad, ya que los plugins se componen de código arbitrario ejecutado en el espacio de memoria del programa principal, con lo que pueden alterar el funcionamiento de este.

Se ha decidido utilizar un sistema de plugins, puesto que el objetivo de flexibilidad es el principal del sistema, además de asumirse que la mayor parte de plugins van a ser desarrollados internamente por la organización y no por terceras partes, con lo que la seguridad no es tan problemática.

Para implementar este sistema, deben analizarse los posibles puntos de extensión por plugins y desarrollar interfaces para cada uno de estos. Esto debe

incluir como mínimo los siguientes aspectos:

- La autenticación de los usuarios, para facilitar la integración con sistemas preexistentes.
- Modificación de los datos y metadatos de los ficheros a sincronizar, para añadir funcionalidad como puede ser el cifrado o la compresión de los datos.
- La interfaz de usuario, de forma que puedan utilizarse distintas interfaces según los gustos o necesidades de los usuarios, además de permitir su uso en entornos sin interfaz gráfica.

3.3. Arquitectura de red

La sincronización de ficheros supone el núcleo de la herramienta y por tanto precisa recibir el mayor detalle en el análisis. Se debe elegir aspectos tales como el mecanismo de detección de cambios en ficheros, el protocolo de red para comunicar estos cambios, la arquitectura a nivel de red del sistema, etc.

En primer lugar, se debe decidir el tipo de arquitectura de red a utilizar, ya que esto condiciona en gran medida otras decisiones del sistema, como el protocolo de red a utilizar o el comportamiento cuando uno de los equipos está fuera de línea. Los elementos imprescindibles en la arquitectura son los equipos cliente, aquellos entre los que se sincronizan los ficheros. Opcionalmente, se puede considerar añadir equipos servidores que faciliten la comunicación entre los clientes, que pueden proporcionar distintos servicios: localización de otros clientes, comportamiento de cliente, cola de mensajes (para evitar perder mensajes entre clientes), etc.

Con esto en mente, se proponen tres arquitecturas de red, esquematizadas en la figura 3.1.

3.3.1. Peer to peer total

No hay ningún equipo servidor. Los datos se intercambian directamente entre los clientes, que se identifican entre ellos por introducción directa de sus direcciones de red o se autodetectan utilizando técnicas de broadcast en la red local. La autenticación se realizaría directamente entre los clientes, probablemente utilizando un secreto compartido.

Tiene la ventaja de no necesitar un servidor, y prácticamente ninguna configuración en caso de utilizarse en una red local con autodetección de otros clientes. Sin embargo, en caso de que ningún otro cliente esté conectado en un momento dado, la sincronización es imposible. Además, si se desea guardar información adicional sobre los ficheros (tal como historial de versiones antiguas), es necesario replicarla en todos los clientes para garantizar la mayor disponibilidad posible de estos datos, lo cuál puede aumentar mucho el tamaño de los datos a almacenar por cada cliente.

3.3.2. Peer to peer con localizador

Existe un equipo servidor cuya única función es autenticar a los clientes que desean acceder a un punto de montaje, e informar de otros clientes conectados.

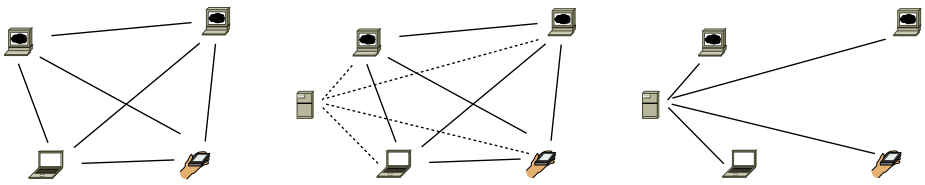
El servidor necesario para este tipo de sistema es muy ligero, con lo que podría incluso implementarse en un cliente con garantías de alta disponibilidad.

Esto facilita la conexión entre cliente en distintas subredes a cambio del coste de precisar de un servidor para facilitar esto, pero por lo demás se comporta como la arquitectura descrita anteriormente y comparte las mismas desventajas sobre la necesidad de tener al menos un cliente conectado para llevar a cabo la sincronización. Además, es necesario destacar que este servidor se convierte en un punto único de fallo a la hora de la autenticación y localización de clientes. Sin embargo, una vez el cliente está conectado al sistema no hay necesidad de contactar con el servidor, con lo que podría seguir funcionando durante largos periodos de tiempo sin este servidor.

3.3.3. Cliente-Servidor

Quizás la arquitectura más tradicional, consiste en un servidor que centraliza toda la operación del sistema y sin el cuál el sistema no puede realizar ni sus tareas más básicas. Este servidor se ocupa de la autenticación, envío y recepción de notificaciones de cambios y datos modificados, además de otras funcionalidades adicionales. Todos los clientes se comunican directamente con el servidor y no entre ellos, con lo que un fallo en este servidor detiene el funcionamiento del sistema inmediatamente, convirtiéndose en un punto único de fallo muy importante.

Sin embargo, las ventajas son aparentes en la riqueza de funcionalidad adicional que permite, por ejemplo: archivo de versiones viejas de los ficheros, simplificación de los clientes (y evitar la necesidad de que alguno de estos estén siempre activos) y simplificación de la resolución de conflictos (el servidor puede decidir cuál es la última versión recibida).



(a) P2P (b) P2P con localizador (c) Cliente-Servidor

Figura 3.1: Arquitecturas de red

3.3.4. Evaluación

La arquitectura peer to peer permite evitar la necesidad de un servidor, ya que los cliente se sincronizan entre ellos, siempre que estén conectados. Sin embargo, para que el sistema funcione adecuadamente es recomendable que alguno de los clientes tenga garantías de disponibilidad, de manera que la sincronización sea posible por haber disponible al menos otro cliente disponible con el que sincronizar los datos. Además es conveniente que esté sea siempre el mismo, puesto que en otro caso se podrían formar grupos de clientes con datos consistentes entre sí pero no con el resto del sistema, por ejemplo: una empresa en

la que los empleados de distintos turnos usaran el sistema para sincronizarse podría acabar con una *islas* de consistencia para cada turno.

Como un cliente con alta disponibilidad se asemeja a un servidor, esta arquitectura pierde su gran ventaja, y dada la complejidad de implementarla y que se ha desechado su principal beneficio, podemos descartarla frente a las otras dos opciones.

Siguiendo la discusión anterior, se debería situar un cliente en el servidor de localización para garantizar la consistencia de los datos en todo el sistema. Sin embargo, al ser una arquitectura peer to peer, los datos se enviarían entre clientes, probablemente saturando el ancho de banda de estos (al tener que enviar varias copias de cada fichero); mientras que en el caso de servidor cada fichero se envía y descarga siempre del servidor, que normalmente tendrá un ancho de banda mucho mayor.

Además, como se comentó anteriormente, en una arquitectura tipo cliente-servidor es más sencillo mantener consistencia (mediante operaciones atómicas en el servidor, frente a una base de datos distribuida en arquitecturas peer to peer) y permite ofrecer servicios avanzados en el servidor, dado que es un ente diferente a los clientes.

Dado que el objetivo principal del proyecto es proveer un sistema flexible, la arquitectura elegida es la de cliente-servidor, que además de permitir la mayor cantidad de funcionalidad avanzada, proporcionará un rendimiento igual o superior a las otras en escenarios típicos y podrá implementarse con menos recursos debido a su sencillez.

3.4. Sincronización entre múltiples usuarios

En la sección anterior se ha elegido una arquitectura cliente-servidor para el sistema. Tradicionalmente, en estas arquitecturas cada cliente se conecta a un servidor que recibe conexiones de varios clientes. Sin embargo, en este sistema un cliente no quedará restringido a conectarse a un único servidor, si no que podrá conectarse a varios simultáneamente, con lo que se permitirá que una única instancia de un cliente pueda sincronizar datos de varios directorios con distintos servidores, a diferencia de las soluciones mencionadas en el estado del arte, enfocadas y restringidas a un único servidor (el del proveedor del servicio).

El objetivo de esto es facilitar la sincronización entre usuarios con datos almacenados en distintos servidores, pudiendo el propietario de los datos en cada servidor dar acceso a otros usuarios a trabajar sobre estos datos para permitir la compartición de datos sin compartir las credenciales de acceso al sistema.

Además, puesto que se pueden sincronizar varios directorios, también deberá poder hacerse esto en un único servidor, por lo que cada servidor deberá ser capaz de almacenar datos del mismo usuarios pero distintos directorios de sincronización. Para esto, se identificará cada directorio compartido del cliente en el servidor con el nombre del usuario y un identificador para el directorio compartido cuyo nombre puede coincidir o no con el nombre del directorio local del cliente.

Por comodidad, a partir de ahora se referenciará al directorio local del cliente como “punto de montaje” y al equivalente en el servidor como “share” o “compartición”, resaltando el hecho de que se permitirá la conexión a shares

de otros clientes siempre y cuando el propietario permita su acceso, siendo esta característica imprescindible para la sincronización de datos entre múltiples usuarios.

3.5. Comunicaciones

3.5.1. Requisitos

Para la comunicación entre las distintas entidades del sistema es necesario el desarrollo de un protocolo de red adecuado al sistema, así como la elección de los protocolos de red subyacentes para completar la pila de protocolos. Para ello, es necesario analizar las necesidades y características de las comunicaciones en el sistema:

- **Comunicación punto a punto:** Ya que se ha elegido una arquitectura de cliente-servidor, la comunicación se desarrolla entre una entidad de cada tipo, con lo que no es necesario comunicación multi-punto.
- **Integridad:** El protocolo se utiliza principalmente para transmitir ficheros y metadatos asociados a ellos, con lo que es necesario asegurar la integridad de los datos para evitar la corrupción de los ficheros.
- **Confidencialidad:** Dado que se va a tratar con datos sensibles del usuario a través de una red (probablemente Internet), es necesario asegurar que estos datos no pueden ser vistos por terceras personas.
- **Distintos tipos de datos:** El protocolo se va a utilizar tanto para enviar notificaciones y órdenes (control) como para el envío de grandes cantidades de datos en bruto, por lo que se debe proveer de una forma de enviar ambos tipos de datos sin que los grandes envíos ralenticen a los comandos.
- **Notificaciones bidireccionales:** Tanto el cliente como el servidor deben ser capaces de notificar al otro cuando algún fichero ha sido modificado, ya sea para enviarlo al resto de clientes o recibirlo. Es preferible que estas notificaciones tengan carácter instantáneo para una mayor fluidez en la sincronización de datos.

3.5.2. Transporte

Dada la necesidad de confidencialidad e integridad, posiblemente la mejor opción como protocolo de nivel de transporte es TLS (Transport Layer Security) [12], el protocolo estándar usado en multitud de servicios seguros como HTTPS, FTPS o XMPP. Dado su ubicuidad, principalmente por su uso en HTTPS, está disponible en la mayoría de equipos de forma estándar, sin necesidad de programar o instalar software adicional.

Para el tratamiento de distintos tipos de datos se debe proporcionar un servicio de multiplexado (enviar datos de varios canales a través de un único canal), siendo la opción más deseable MTLS (Multiplexed Transport Layer Security) [13] por ser un estándar existente y usar TLS para proporcional los servicios de seguridad. Sin embargo, el documento enviado al IETF sigue en fase de borrador, sin haber actividad desde finales de 2009. Dado que el protocolo no está estandarizado ni parece que vaya a serlo en el futuro próximo, se decide diseñar un

protocolo similar a este (pero simplificando ciertos aspectos innecesarios en este proyecto).

3.5.3. Aplicación

Para la parte del protocolo específica a la aplicación ha de decidirse la forma de representar los datos, y si se va a utilizar algún protocolo de nivel de aplicación para transportarlos o se va a emplear directamente el de nivel de transporte. Habrá que tener en cuenta protocolos tanto para las notificaciones como para la transferencia de datos, que tengan todas las funcionalidades deseadas y no sean muy complejos.

Protocolo

Se puede considerar el empleo de protocolos ya existentes y altamente usados como HTTP y FTP para las notificaciones y transferencias de ficheros respectivamente. Sin embargo ambos proveen muchas mas funcionalidades (y por tanto complejidad) de las necesarias en este dominio de aplicación, además de tener que modificarlos levemente para funcionar sobre el protocolo de multiplexado. A pesar de esto se podría considerar utilizar estos protocolos porque los sistemas de seguridad suelen estar configurados para permitir su uso, si no fuera por una deficiencia grave en HTTP para el uso que se le dará en el proyecto, la incapacidad de establecer comunicaciones bidireccionales (en las que ambos extremos puedan enviar cuando lo deseen) a través de las que enviar las notificaciones.

Otra opción es emplear un sistema de RPCs (Remote procedure calls) para las notificaciones, pero como en el caso de HTTP, el servidor no puede enviar peticiones al cliente directamente. Tanto aquí como en el caso anterior se podrían establecer mecanismos de *polling* en los que el cliente solicitara periódicamente al servidor las notificaciones, pero esto aumentaría el tiempo de respuesta a notificaciones enviadas del servidor al cliente.

Una última opción a considerar son protocolos de mensajería instantánea como XMPP para el envío de las notificaciones. Estos protocolos proveen todas las características necesarias para la aplicación, incluida la capacidad de envío de notificaciones por ambos extremos. Sin embargo, puesto que estos protocolos están pensados para comunicaciones entre gran número de personas, añaden a los mensajes datos como el destinatario o el remitente de cada mensaje, datos totalmente innecesarios para este proyectos puesto que cada canal de comunicación se establece únicamente entre un cliente y un servidor. Toda esta información adicional no haría más que aumentar artificialmente el tamaño de los mensajes, reduciéndose el rendimiento del sistema.

Dado que no hay un protocolo existente con las características deseadas, se recurre a diseñar uno específico para esta aplicación, que combinará aspectos de los protocolos de RPC o HTTP (el cliente hace peticiones que el servidor responde) y de los protocolos de mensajería instantánea (la capacidad de enviar mensajes de ambos extremos de la comunicación).

Representación de los datos

Además de un protocolo sobre el que transmitir los datos, es necesario establecer la forma de representarlos de forma que ocupen el menor espacio posible

(para maximizar el rendimiento) y que permitan la representación de toda la información sin pérdidas. Algunas alternativas populares para la representación de datos son XML, JSON o los sistemas de serialización binaria, todas disponibles para multitud de plataformas y lenguajes de programación.

De las tres opciones, la más adecuada es la serialización de objetos, puesto que típicamente es la que ocupa menos tamaño (mejorando el rendimiento de la red) y se codifica más rápidamente (con menos gasto de CPU y memoria) al ser una representación nativa. Puesto que los datos enviados no van a ser almacenados ni tratados por herramientas externas al sistema, es seguro utilizar esta alternativa, ya que los principales beneficios de XML y JSON son la facilidad de tratamiento de datos por diferentes herramientas, lo que consiguen a costa de un mayor tamaño y complejidad.

Por todo esto, se decide utilizar una herramienta de serialización de objetos en formato binario, ya sea la incluida con el lenguaje de programación u otra que se implementa mediante el uso de librerías o manualmente, siempre que exista documentación abundante sobre el formato de datos (para evitar el *lock-in* en la plataforma) y esta permita representar todos los datos necesarios por la aplicación, entre los que destaca el de representar el texto en alguna forma de Unicode para permitir cualquier tipo de carácter tanto en los datos, como en sus metadatos (incluyendo el nombre de los ficheros).

3.6. Consistencia

Como en todo sistema de tratamiento de datos, es imprescindible considerar los problemas de la consistencia del sistema de ficheros. Esto es especialmente importante en un sistema distribuido, en el que los datos pueden ser modificados simultáneamente en múltiples máquinas, con el consiguiente problema a la hora de sincronizar los datos.

No sólo pueden crearse conflictos por modificación simultánea, si no que pueden producirse en cualquiera de las cuatro operaciones principales sobre un sistema de ficheros: creación, borrado, modificación y cambio de nombre (incluyendo cambios de ruta o mover ficheros). Por ejemplo, dos usuarios podrían crear simultáneamente un fichero con mismo nombre y distintos datos, o un usuario puede borrar un fichero a la vez que otro lo modifica.

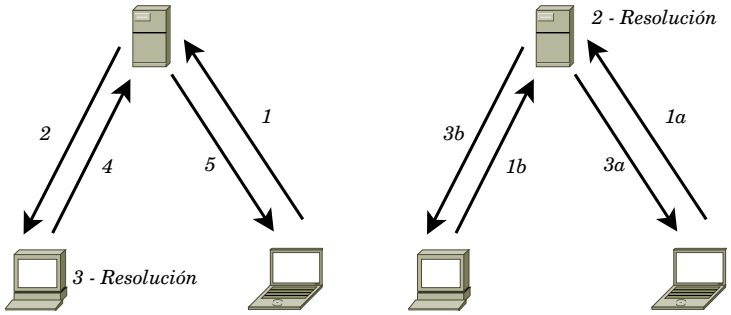
Este proyecto no trata de mecanismos de resolución de conflictos, como puede ser un intento de unión de cambios, dejándose esta funcionalidad al desarrollador de extensiones del programa. Así, es necesario permitir que los plugins sean capaces de acceder al mecanismo de resolución de conflictos, para poder modificar y/o ampliar su comportamiento.

Estos plugins podrían ser implementados en los clientes (con lo cuál cada cliente podría tener distintos criterios para tratar los conflictos) o en el servidor (comportamiento uniforme para cada punto de montaje).

Las ventajas de su implementación en los clientes es la descarga de trabajo del servidor y una mayor flexibilidad a la hora de crear comportamientos, permitiendo fácilmente asimetrías entre los distintos clientes.

La implementación en el servidor facilita la uniformidad, descargando trabajo de los clientes pero añadiéndosela al servidor. El coste en términos de red es aproximadamente el mismo como se puede comprobar en la figura 3.2, dos operaciones de subida y dos de descarga, siendo sin embargo el tiempo apro-

ximadamente el doble en el caso de la resolución en el cliente por no poder paralelizarse dicho tráfico.



(a) Resolución en cliente (b) Resolución en servidor

Figura 3.2: Tráfico de red en los distintos modelos de resolución de conflictos

Siguiendo el objetivo principal del proyecto, se escoge el modelo de resolución de conflictos en el cliente, por permitir la mayor personalización posible del sistema, además de por evitar posible problemas de rendimiento en el servidor en caso de resolver los conflictos en este, pudiendo causar denegaciones de servicio.

Esto permite utilizar pocos recursos de servidor para un gran número de cliente, puesto que se distribuye la carga de trabajo entre todos los clientes del sistema, lo que es apropiado para casos de uso con gran números de estaciones de trabajo medianamente potentes (cualquier ordenador de sobremesa o portátil).

Sin embargo, para el uso en dispositivos móviles esto podría ser un problema, por la menor capacidad de estos para tratar con grandes ficheros, pero el se estima que el escenario en el que estos dispositivos se sincronizan bidireccionalmente con el resto del sistema es mínimo y aún siendo esta la situación, podrían implementar políticas de consistencia simples, en las que simplemente descartan los cambios de uno de los dos dispositivos, por ejemplo, basándose en la fecha de última modificación del fichero.

3.6.1. Política por defecto

Para garantizar el funcionamiento del sistema, se debe proporcionar una política de resolución de conflictos adecuada por defecto, de forma que en la ausencia de un plugin de este tipo el sistema siga siendo usable. Para garantizar la mejor experiencia de uso posible se ha construido un pequeño cuestionario con los posibles casos de aplicación de este algoritmo y se han distribuido a varios usuarios para recoger sus respuestas. El cuestionario consistía en la tabla 3.1, junto con unas instrucciones indicando la posibilidad de utilizar acciones fueras de las sugeridas en esta, o ejecutar una acción dependiendo de cualquier otra condición dependiente de los datos o metadatos del fichero. No se incluye la acción de movimiento/cambio de nombre por poder considerarse esta como un borrado y una creación con los mismos datos, de forma que se utilizará el mismo comportamiento que se aplicaría al conjunto de dos operaciones consecutivas de borrado y creación.

En la tabla 3.2 se muestran las respuestas de cada uno de los entrevistados al

Versión cliente	**Versión servidor**	**Evento**	**Escenarios típicos**	**Acciones posibles**	**Acción**
No existe	2	Creado	1ª sync, fichero borrado localmente	Descargar, Borrar en servidor	
No existe (existía)	2	Creado	Fichero borrado localmente	Descargar, Borrar en servidor	
Nuevo	No existe	Borrado	1ª sync, fichero creado localmente	Subir, Borrar en local	
1	No existe	Borrado	Fichero borrado en otro equipo	Subir, Borrar en local	
1 (mod)	No existe	Borrado	Fichero borrado en otro equipo y cambios locales	Subir, Borrar en local	
Nuevo	2	Cambio	1ª sync, fichero borrado y recreado en local	Descargar, Subir	
1	2	Cambio	Fichero actualizado en otro cliente	Descargar, Subir	
1 (mod)	2	Cambio	Fichero actualizado en 2 clientes (este y otro)	Descargar, Subir	
2 (mod)	2	Cambio	Fichero actualizado en local	Descargar, Subir	

Tabla 3.1: Cuestionario de resolución de conflictos

Versión cliente	**Versión servidor**	**Evento**	**1**	**2**	**3**
No existe	2	Creado	**Descargar**	**Descargar**	**Descargar**
No existe (existía)	2	Creado	**Borrar en servidor**	**Borrar en servidor**	**Borrar en servidor**
Nuevo	No existe	Borrado	**Subir**	**Subir**	**Subir**
1	No existe	Borrado	**Borrar en local**	**Borrar en local**	**Borrar en local**
1 (mod)	No existe	Borrado	Fecha mod.	Borrar en local	**Subir**
Nuevo	2	Cambio	**Fecha mod.**	Descargar	Descargar
1	2	Cambio	**Descargar**	**Descargar**	**Descargar**
1 (mod)	2	Cambio	**Fecha mod.**	**Fecha mod.**	Subir
2 (mod)	2	Cambio	**Subir**	**Subir**	**Subir**

Tabla 3.2: Resultados de cuestionario de resolución de conflictos

cuestionario, así como la decisión final tomada (resaltada en negrita). Se omiten las columnas informativas de escenario típico y acciones posibles por brevedad. "Fecha mod" significa comparar la fecha de modificación de los ficheros y conservar la copia más nueva: si la copia más reciente es la del servidor, descargar, en caso contrario subir (o borrar en caso de que el fichero se haya borrado localmente). En la mayoría de casos, los usuarios consultados estaban de acuerdo, en los casos que diferían se tomó una única decisión:

- **Borrado de un fichero modificado localmente:** El caso más discutido. Se decide subir el fichero incondicionalmente, porque en el peor caso, se recreará un fichero borrado y deberá volverse a borrar. En las otras opciones, existe el riesgo de pérdida de datos, ya que puede decidirse el borrado en el servidor.
- **Modificación de un fichero desconocido:** La opción elegida por dos usuarios es descargar el fichero, seguramente por no darse cuenta de que un fichero desconocido puede existir localmente. Se decide consultar la fecha de modificación del fichero del servidor y la local, y conservar el fichero más moderno (puede implicar subir o descargar el fichero).
- **Modificación de un fichero modificado localmente:** Uno de los usuarios sugiere subir los cambios directamente, mientras que los otros dos prefieren conservar el fichero más moderno. Se elige la segunda opción ya que es probablemente la más adecuada en el caso general. Este es el caso principal que promueve la creación de plugins, ya que dependiendo del tipo de fichero, se pueden intentar unir los cambios, etc.

3.7. Almacenamiento de los datos

Un problema común entre los sistemas de este tipo es el almacenamiento de los datos y metadatos de los ficheros y directorios a sincronizar ya que la opción obvia, almacenar los datos en el servidor como en el cliente, no permite añadir funcionalidad adicional fácilmente además de ser las consultas muy lentas por tener que recorrer el árbol de inodos para cualquier operación.

Es por esto, que se decide utilizar una base de datos como sistema de almacenamiento de, al menos, los metadatos de los objetos sincronizados. Además de ser mucho más eficiente, esto permite el almacenamiento de metadatos adicionales necesarios para el funcionamiento del sistema, como el dueño de los ficheros, los permisos (que no podrían almacenarse directamente en el inodo puesto que esto podría impedir el acceso del servidor al fichero) o cualquier otra información adicional.

Los datos también pueden almacenarse en la base de datos, pero el gran tamaño de estos hace inadecuada esta solución, al menos en la misma base de datos de los metadatos. Esto es porque lo más adecuado para el almacenamiento de los metadatos es un motor de bases de datos relacionales que garantiza la integridad de los datos y proporciona una gran velocidad de acceso a estos mediante índices, pero a su vez son muy ineficientes para grandes cantidades de datos.

Existen soluciones como los gestores de documentación que serían capaces de almacenar la gran cantidad de datos requerida de manera eficiente, pero dato que

de todos modos no existiría integración directa con los metadatos almacenados en la base de datos, y que no es necesario realizar operaciones sobre estos datos, el almacenamiento en un sistema de ficheros es adecuado.

Para almacenarlo en un sistema de ficheros es necesario proporcionar un sistema para identificar los datos y relacionarlos con los metadatos. Algunas opciones es utilizar el nombre del fichero real y modificarlo si el cliente cambia el nombre, que podría ocasionar problemas en caso de cambios de nombres simultáneos. También se podría utilizar un identificador único, secuencial o aleatorio y añadir este campo a la base de datos de metadatos.

Sin embargo, la solución elegida utiliza algoritmos de hash o resumen, que toman los datos del fichero y generan una secuencia de bits resumida que identifica de forma "única" a los datos. Evidentemente, como el resumen es de menor longitud que los datos por lo general, el identificador no puede ser único, sin embargo, los algoritmos de hash criptográficamente seguros hacen prácticamente imposible hallar ficheros con mismo identificador y distintos datos. Asumiendo que todos los seres humanos crearan un fichero cada milisegundo, se tardarían del orden de trillones (10^{18}) de años en encontrar dos ficheros con mismo resumen de 128 bits.

Esto tiene la ventaja de no tener que generar un identificador, si no que se puede utilizar el resumen de los datos como este, aprovechando además las características de las funciones resumen para garantizar la integridad de los datos: si el resumen de los datos coincide, los datos deben coincidir.

3.8. Plataforma de desarrollo

Otra decisión importante es la elección de la plataforma software sobre la que desarrollar el proyecto. Los requisitos y características deseables para esta plataforma incluyen:

- Multiplataforma, al menos entre los tres sistemas operativos de escritorio principales.
- Facilidad de creación de plugins y cargado dinámico de estos.
- Fácil integración con las APIs del sistema operativo, para poder implementar las operaciones a bajo nivel necesarias para la supervisión del sistema de ficheros.
- Conjunto completo de herramientas, desde diseño de interfaces de usuario hasta comunicación por red.
- API de serialización bien documentada (ver sección 3.5.3).

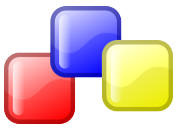
3.8.1. Alternativas

Las opciones consideradas que mejor cumplen con los requisitos aquí descritos son Java [14] y C++ con la librería Qt [15]. Otras de las opciones analizadas fueron la plataforma .Net[16], descartada por la dificultad de uso en plataformas no Windows (debiéndose recurrir a proyectos no oficiales como es Mono [17]) o C++ con wxWidgets [18], descartado por su similaridad a la alternativa con QT, además de no incluir serialización binaria de objetos.

Java es un lenguaje de programación y plataforma de desarrollo multipropósito, especialmente utilizado para la creación de aplicaciones de escritorio(Java SE) y móviles (Java ME), además de aplicaciones web(Java EE). Incluye en el núcleo herramientas para casi cualquier objetivo, totalmente abstraídas del sistema operativo subyacente a través de la máquina virtual de Java (JVM). Como lenguaje de alto nivel, incluye muchas facilidades para el programador, lo que facilita el desarrollo, pero dificulta el acceso a APIs de bajo nivel, como las de los sistemas operativos.

Qt es un framework para C++ (aunque también soporta otra docena de lenguajes como Python o Java) dirigido al desarrollo de aplicaciones completas de escritorio y móviles. Incluye abstracciones sobre librerías gráficas con aspecto nativo en cada plataforma soportada, de red, bases de datos, OpenGL, etc.

.Net es un framework para el desarrollo de aplicaciones en Windows (aunque existe soporte extraoficial para otros sistemas operativos a través del proyecto Mono). Es bastante similar a Java, en el aspecto de integrar una gran librería de clases de alto nivel y ejecutarse sobre una máquina virtual, aunque permite la programación en múltiples lenguajes como C# o VB.net.



wxWidgets es un framework para C++ y Python cuyo principal objetivo es proporcionar una abstracción sobre las librerías gráficas de diversos sistemas operativos, de forma que los programas resultantes tengan aspecto completamente nativo sobre las arquitecturas soportadas. También provee funcionalidad multiplataforma para el acceso a al red, al sistema de ficheros, etc.

3.8.2. Comparación

Tanto Java como Qt soportan los sistemas operativos de escritorio principales: Windows, Mac OS y distintas variedades de Unix, incluyendo Linux o la familia BSD. Además ambos puede ejecutarse (con limitaciones) en la mayor parte de dispositivos móviles, siendo el soporte de Java más maduro en este aspecto puesto que el soporte de Qt en alguna de estas plataformas como iPhone o Android es todavía experimental.

Para la creación de plugins, Java utiliza carga dinámica de ficheros .jar, que contienen código Java compilado. Qt utiliza un sistema más básico, carga

de librerías dinámicas (.dll en Windows, o .so en muchos Unix), que pueden escribirse en cualquier lenguaje de programación con un compilador para el sistema operativo objetivo. Además Qt permite el uso de un lenguaje de script propio, QtScript, muy similar a JavaScript para desarrollar extensiones.

Otro punto importante es el acceso a las APIs de los distintos sistemas operativos para la monitorización de cambios en los sistemas de ficheros. En el caso de los sistemas operativo de escritorio, estás están en forma de cabeceras de C/C++, con lo que su uso desde C++ es trivial. Java proporciona un sistema para acceder a estas (JNI: Java Native Interface) pero esto requiere la creación de unas interfaces Java-C que, aunque el proceso está semi-automatizado, requiere un trabajo adicional.

A nivel de madurez y completitud de las plataformas, ambas son muy similares en lo que respecta a este proyecto, proporcionando gestión de ficheros, buffers de memorias, conexiones de red, etc. Una ventaja de Java respecto a C++ es la gestión automática de memoria, lo que es especialmente importante al trabajar con tantos buffers de lectura y escritura, dado que se pueden producir fugas de memoria fácilmente. Sin embargo, Qt añade un recolector de basura muy básico a C++ que facilita esta gestión, aunque sigue siendo necesario cierto cuidado.

Finalmente, ambas plataformas ofrecen APIs de serialización de objetos adecuadamente documentadas [19, 20]. Java incluye más información en su formato, añadiendo datos como el nombre de las clases y atributos, mientras que Qt se limita a añadirlos siempre en el mismo orden. Esto hace el método de Java más robusto y compatible entre versiones, mientras que el formato de Qt es más ligero.

3.8.3. Decisión

Finalmente, se decide desarrollar la aplicación en C++ con el framework Qt, debido principalmente a su mayor selección de lenguajes a la hora de desarrollar extensiones, lo que facilitará la creación de estos por terceras partes. También influye en la decisión su mayor cercanía a las APIs del sistema operativo y el menor tamaño de su formato de serialización, lo que permitirá que los paquetes enviados por la red sean de un tamaño algo menor.

Introducción a Qt

Dado que se va a trabajar con Qt, se utilizarán algunos de los términos particulares de este framework, con lo que conviene conocerlos para comprender correctamente el diseño.

La principal característica diferenciadora de Qt respecto a otros frameworks es su sistema de señales y slots, documentado en [21]. Este sistema está diseñado con la idea de promover el bajo acoplamiento entre diversos componentes del programa, aunque puede utilizarse con otros fines.

La idea básica es que cada clase (que herede de `QObject`) tiene una serie de *señales* que se `emiten` a manera de eventos cuando se actúa sobre un componente. Algunos ejemplos son la señal `clicked()` de los botones de la interfaz o `readyRead()` emitida por los dispositivos de entrada/salida cuando tiene disponibles más datos para leer.

En contrapartida, existen los *slots*, que no se diferencian de métodos normales, a excepción de su posibilidad de *conectarlos* a señales. Al establecer una conexión entre una señal y un slot, el sistema de metaobjetos de Qt se ocupará de ejecutar el método slot cada vez que se emita la señal.

Una señal puede ser conectada a varios slots (que se ejecutarán secuencialmente) y un slot puede ser conectado a varias señales (se ejecutará cada vez que se emita cualquiera de las señales) como se puede ver en la figura 3.3.

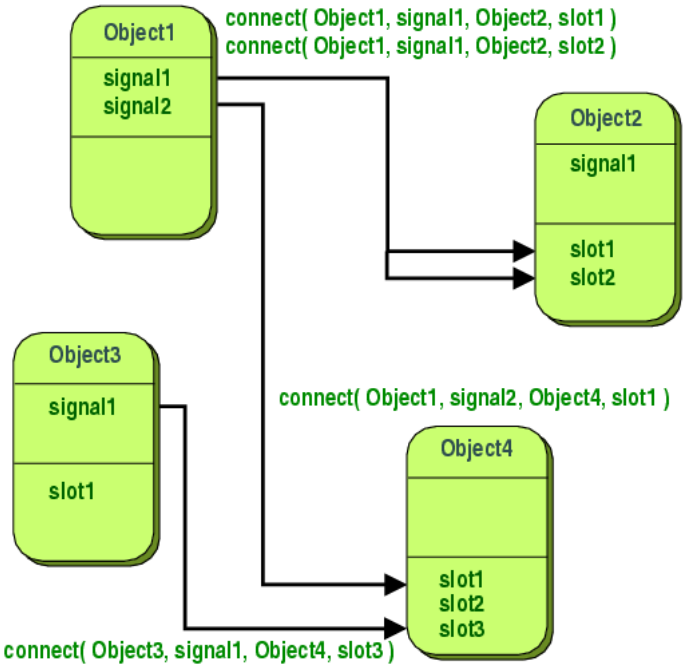


Figura 3.3: Señales y slots

Tanto las señales como los slots admiten parámetros, que se pueden utilizar para transmitir información detallada sobre el evento, por ejemplo, la señal `stateChanged()` de `QAbstractSocket` tiene un parámetro que indica el nuevo estado del socket.

Todo el mecanismo de señales y slots es seguro ante el polimorfismo, debido al sistema de metaobjetos de Qt que utiliza macros y un compilador especial (moc) para marcar la clase real de cada objeto. En este aspecto es similar al mecanismo estándar de C++: RTTI (Real Time Type Information) pero más detallado, incluyendo capacidades potentes de introspección o el mencionado sistema de señales y slots.

Por último, pueden realizarse conexiones entre señales y slots de forma segura a través de distintos hilos de ejecución. En este caso, Qt lo detectará automáticamente y creará una conexión del tipo `Qt::QueuedConnection` que utiliza una cola de procesado de eventos en el hilo destino, a diferencia del tipo normal (`Qt::DirectConnection`) que ejecuta el método slot directamente. Durante el desarrollo de este proyecto se utiliza esta funcionalidad para gestionar la comunicación entre hilos sin tener que recurrir a una implementación manual, aprovechando las conexiones encoladas ya proporcionadas por el framework.

Parte III

Diseño del prototipo

Capítulo 4

Diseño del protocolo

El protocolo de red es el componente del sistema que permite la interoperabilidad entre el cliente y el servidor. Para ello se define el formato de transmisión por red básico y de cada uno de los comandos enviados, así como de las respuestas.

En realidad, debido a las distintas necesidades del sistema, se utilizan varios protocolos a distintos niveles, utilizando el modelo de la pila de protocolos, tal como el modelo OSI o TCP/IP. La pila de protocolos completa se muestra en la figura 4.1.

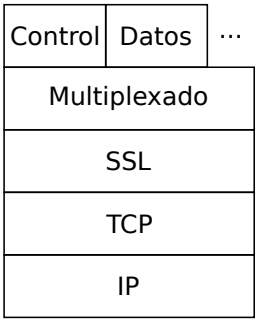


Figura 4.1: Pila de protocolos de red

4.1. Multiplexado

En primer lugar, como se comentó en la subsección 3.5.2 se diseñará un protocolo para el multiplexado de distintos canales de datos, de forma que se puedan enviar datos distintos (principalmente control y transmisión de ficheros) a través de la misma conexión TCP.

4.1.1. Formato del paquete

Para diferenciar los canales son imprescindibles dos cosas: un indicador del canal al que pertenecen los datos y una forma de delimitar los datos de cada canal. Para esto se pueden emplear dos técnicas principales: el uso de un carácter

especial de delimitación o la utilización de marcas de longitud antes de cada envío. Se elige la segunda opción, puesto que la primera es más difícil de implementar en un protocolo binario (en el que puede aparecer cualquier secuencia de bytes).

Para ello, simplemente se envían fragmentos de la información del canal prefijados por el número de canal (2 bytes) y la longitud de los datos enviados (2 bytes, no incluye la longitud de las cabeceras) como se muestra en la figura 4.2. Se ha decidido utilizar 2 bytes para el número de canal (65536 posibilidades) para permitir que distintos plugins puedan utilizar sus propios canales de datos para su uso propio, de forma que puedan decidir su propio formato de comunicación sin tener que utilizar el del núcleo del programa.

A pesar de considerarse que un sólo byte podría ser suficiente para el número de canal (256 canales), se decidió utilizar un byte adicional para no limitar futuras expansiones del protocolo, y para poder dividir el espacio de identificadores en distintas categorías para indicar la prioridad, tal y como se discute en detalle en la subsección 4.1.2.

Respecto a la longitud, 2 bytes equivalen a 64KiB, lo que es significativamente superior al MTU de la mayoría de redes (por ejemplo, 1500 bytes para Ethernet), por lo que debería ser suficiente. El tamaño menos es de 1 byte de longitud o 256 de datos, claramente insuficiente por ser menor que el MTU. Utilizar un tamaño mayor aumentaría el overhead del protocolo en la mayoría de casos, puesto que la mayoría de envíos no superarán dichos 16KiB.

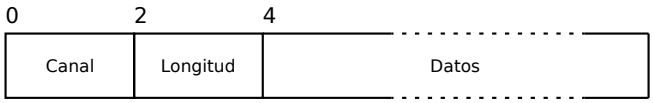


Figura 4.2: Protocolo de multiplexado

4.1.2. Gestión de canales

Además de la capacidad de enviar datos de distintos canales, es necesario decidir cuándo enviar datos de cada canal. Para esto, primero hay que tener en cuenta los distintos usos de cada canal, entre los cuáles podemos distinguir dos tipos principales: de control y de transferencia de datos.

Los canales de control necesitan menor latencia y normalmente poco ancho de banda, mientras que los de datos necesitan gran ancho de banda pero una latencia mayor no es problemática. Por tanto, podemos asignar una mayor prioridad a los canales de control, de manera que si hay datos del canal de control listos para enviar, se envían estos y en otro caso se envían los del canal de datos.

Con el añadido de las extensiones, se genera el dilema de distribuir los recursos entre varios canales del mismo tipo, para lo cuál se adopta la política de dar prioridad al canal de control principal, ya que es necesario para el buen funcionamiento del sistema al ser el encargado de gestionar la conexión. El resto de canales de control de las extensiones se distribuyen el ancho de banda disponible utilizando una política de "round-robin".

Se aplican políticas similares a la transmisión de datos, priorizando el tráfico del núcleo del sistema puesto que una parada en este puede retrasar la sincro-

nización de ficheros, afectando al canal de control principal, y utilizando una política "round-robin" para el resto de canales.

Aplicando todas las políticas, se forman cuatro niveles de prioridad, dos de ellos reservados para el sistema y otros dos compartidos por todos los plugins, tal y como se muestra en la tabla 4.1. Se utiliza el número de canal para indicar el tipo de prioridad, de forma que no sea necesario mantener una estructura de datos adicional. Para ello, se reservan los primeros 1024 canales para posibles futuros usos del sistema, y se divide el resto de identificadores en dos mitades, para canales de control y de datos de plugins.

Prioridad	**Números de canal**	**Política**	**Uso**
1	0	N/A	Control del sistema
2	1024 a 33279	Round-robin	Control de plugins
3	1	N/A	Datos del sistema
4	33280 a 65535	Round-robin	Datos de plugins

Tabla 4.1: Prioridades de envío del multiplexador

4.2. Protocolo de control

El protocolo principal de la aplicación es el de control, encargado de la conexión, apertura de *shares*, notificación de cambios, control de otros canales, etc.

Para ello, se ha decidido emplear un protocolo simple en el que el cliente envía comandos (representados por un número) y el servidor envía las respuestas, permitiendo además que el servidor envíe notificaciones cuando lo precise. Cabe destacar que las notificaciones enviadas del servidor al cliente no tienen respuestas, ya que el cliente puede optar por ignorarlas o retrasar la acción en función de su política de resolución de cambios o la saturación actual de su cola de sincronización.

El protocolo necesita como mínimo campos para enviar el comando y sus parámetros. Se ha decidido añadir dos campos adicionales para facilitar y hacer más robusto el funcionamiento del protocolo:

- **Número de secuencia:** Un número enviado por el cliente con cada petición, que permite identificar las respuestas del servidor aunque no estén ordenadas. Esto permite que el servidor procese sus mensajes desordenadamente, de manera que las peticiones rápidas pueden responderse instantáneamente, sin esperar a otra peticiones más lentas enviadas antes.

- **Longitud de mensaje:** El tamaño en bytes del mensaje, sin incluir las cabeceras. Esto permite evitar problemas de sincronización en los casos de mensajes malformados, que no puedan leerse correctamente por el receptor. En caso de hallar tal mensaje, simplemente se ignora el número de bytes indicado por este campo, para llegar al siguiente mensaje.

El formato básico de mensaje se muestra en la figura 4.3. El número de secuencia en las respuestas del servidor al cliente será el mismo que en la petición correspondiente, mientras que para las notificaciones que no precisan de

respuesta se utilizará un valor reservado (65535, el valor máximo del campo) que no se utilizará para peticiones normales.

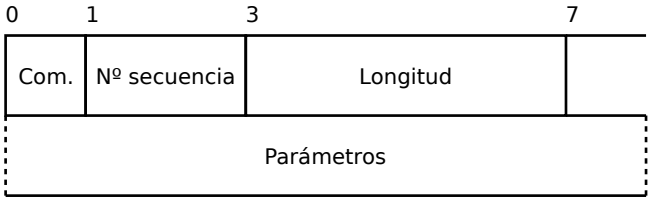


Figura 4.3: Protocolo de control

4.2.1. Representación de los datos

Antes de detallar los comandos, es importante definir el formato binario que tomara cada tipo de datos enviado por la red. Para los tipos básicos, se utilizará el formato estándar de Qt, tal y como se detalla en [20]. Para los tipos propios de la aplicación, se definirá su formato de envío de forma similar al sistema utilizado por Qt, aprovechando en caso de que sea posible sus tipos básicos.

Los tipos de datos utilizados en los parámetros son:

- **Enteros:** 1, 2, 4 u 8 bytes en binario natural (unsigned) o complemento a 2 (signed). Se utiliza la nomenclatura de Qt, que se compone de un prefijo: quint para enteros sin signo y quint para enteros con signo seguido del número de bits que ocupa: 8, 16, 32, 64.
- **QString:** Cadena de caracteres. Longitud (quint32) y datos en UTF-16.
- **QDateTime:** Fecha y hora. Día juliano (quint32), msecs desde medianoche (quint32) y zona horaria (quint8).
- **QByteArray:** Array de bytes (datos). Longitud (quint32) y datos.
- **QList:** Lista de cualquier tipo. Longitud (quint32) y elementos.
- **CommonMetadata:** Metadatos de un fichero o directorio. Tipo (quint8; fichero = 0, directorio = 1), Nombre (QString), Versión (quint32), última modificación (QDateTime).
- **FileMetadata:** Metadatos de un fichero. Los mismos que CommonMetadata más el tamaño (quint64) y el hash de sus datos (QByteArray).

Se ha decidido usar campos no menores de byte, dado que el flujo de TCP, el protocolo de multiplexado y el sistema de serialización de Qt funcionan byte a byte. Las ganancias no serían muy grandes en la mayoría de casos (o incluso se producirían pérdidas, al tener que indicar el tamaño en bits en vez de bytes, lo que haría que el campo de longitud fuera más largo), añadiendo bastante complejidad.

Fechas de modificación y relojes del sistema

Entre los tipos de datos a enviar, se encuentran las fechas de modificación de ficheros en distintos equipos, puesto que se desea que los algoritmos de resolución de conflictos puedan acceder a esta información para poder implementar la política por defecto definida en la subsección 3.6.1.

Sin embargo, para que estas fechas tengan algún valor, deben poder ser comparables entre sí, es decir, que eventos posteriores produzcan fechas posteriores y viceversa. En un sólo equipo esto se cumple siempre, pero en un sistema distribuido como es el objeto del proyecto, es posible (y de hecho, lo más común) que los relojes de los componentes del sistema tengan distinta fecha y hora.

Por todo esto, es necesario sincronizar los relojes de los equipos. Esta sincronización no debería afectar al reloj del sistema, puesto que el usuario puede tener razones para no desear utilizar la misma hora en todos los equipos, incluso obviando las diferencias de zona horaria. Para evitar esto, el programa cliente puede mantener un reloj virtual que esté sincronizado con el resto de clientes y con el servidor, simplemente mediante la aplicación de un desvío sobre el reloj del sistema.

Existen varios protocolos para la sincronización horaria, entre los que destaca NTP (Network Time Protocol) [22], que podría utilizarse como parte del sistema, no sin antes adaptar las implementaciones de este, pues suelen estar diseñadas para modificar directamente el reloj del sistema.

Además, NTP provee gran precisión en la sincronización por medio de medidas de latencia en la comunicación, que en el caso de este sistema no es necesario por los usos que se le da a la hora: fechas de modificación de ficheros. En este dominio, una diferencia de unos milisegundos, o incluso de un par de segundos es a menudo irrelevante: si un mismo ficheros es modificado en dos equipos aproximadamente a la vez, una diferencia de unos segundos no supone una diferencia importante a la hora de decidir qué versión conservar (deberían aplicarse técnicas de fusión de ficheros).

Por todo esto, se puede diseñar un sistema muy sencillo de sincronización de la hora consistente en el envío periódico de la hora del servidor a los clientes, de manera que estos ajusten sus relojes virtuales para que coincidan con los del servidor. La periodicidad es necesaria para reaccionar rápidamente a cambios en el reloj local sin necesidad de incorporar un temporizados que incremente la hora virtual cada segundo.

4.2.2. Comandos

Finalmente, se define cada uno de los comandos del protocolo, así como los parámetros necesarios en cada uno de ellos. En primer lugar se describen las posibles respuestas del servidor a los comandos con su interpretación, dado que muchas de estas son comunes a varios comandos. A continuación se describe cada comando, indicando en caso de ser necesario, interpretaciones o características especiales de las respuestas.

Respuestas

Las respuestas consisten simplemente en un código numérico, sin parámetros que indican el resultado de una operación. Para facilitar el análisis de tráfico,

los errores comienzan en el identificador 200 y la respuesta afirmativa tiene el código 150, a diferencia de los comandos que se numeran comenzando desde 0. El listado de respuestas se encuentra en la tabla 4.2.

Id	Nombre	Interpretación
150	Ok	Operación completada correctamente
200	ErrorLogin	Es necesario loguearse para realizar la operación
201	ErrorNotExists	El objeto (fichero, datos, etc.) pedido no existe
202	ErrorAlreadyOpen	No se puede abrir un share, ya abierto
203	ErrorAlreadyExists	El objeto (fichero, datos, etc.) ya existe, no se puede volver a crear
204	ErrorPermissions	No se disponen de permisos para completar la operación
205	ErrorBusy	El servidor está ocupado con otra operación igual (subidas y descargas)
206	ErrorServer	Error temporal en el servidor
207	ErrorNoData	No existen los nuevos datos de un fichero

Tabla 4.2: Respuestas del servidor a los comandos

En los comandos descritos a continuación, no se incluyen como respuestas posibles dos que siempre pueden producirse: `ErrorServer` en caso de un error temporal del servidor (por ejemplo: problemas de conexión a la base de datos) y `ErrorLogin` en caso de intentar ejecutar cualquier comando distinto de `Login` antes de haberse logueado.

Autenticación: Login (0)

Autentica al usuario en el sistema. Dado que la conexión se realiza a través de un canal seguro (TLS), se puede enviar la contraseña en plano sin problemas de seguridad. Sólo es necesario este dato además del nombre de usuario para iniciar sesión. El servidor puede responder con "Ok" o "ErrorLogin" para indicar éxito o fracaso de la operación respectivamente. Los parámetros se detalla en la tabla 4.3.

Identificador	0	**Nombre**	Login
Parámetros		**Posibles respuestas**	
1. Usuario (QString) 2. Contraseña (QString)		▪ Ok ▪ ErrorLogin	

Tabla 4.3: Comando Login

Apertura de shares: OpenShare (1) e InitShare (2)

`OpenShare` e `InitShare` trabajan conjuntamente para abrir un directorio compartido del servidor. La división es motivada por el alto coste del comando `InitShare` tanto localmente (escaneo completo del punto de montaje) como en

la comunicación (envío de todos los metadatos de la compartición), que podría ser rechazado simplemente por un problema de permisos. Al ejecutarse en dos partes, se garantiza que no se realizará el escaneo del sistema de ficheros ni el envío de los metadatos si la operación no puede tener éxito de manera previsible, es decir, que se haya confirmado la existencia y la capacidad de acceso al share en cuestión.

`OpenShare` (ver tabla 4.4 se ejecuta primero y verifica los permisos de acceso del usuario, etc. Este devuelve un identificador de share, que se utiliza para identificarla en todos los comandos y notificaciones siguientes.

A continuación, se ejecuta `InitShare` que envía una lista del estado del directorio (ficheros existentes, metadatos, etc.) en el cliente, de forma que el servidor puede enviar las notificaciones de cambio correspondientes a las modificaciones realizadas desde la última conexión del cliente. Esto permite que un solo fragmento de código procese los cambios tanto durante la sincronización inicial como durante el funcionamiento normal del sistema, lo que evita inconsistencias entre estos dos momentos. Los detalles de este parámetro se encuentran en la tabla 4.5.

En este mensaje se utiliza un tipo de datos especial, el árbol de ficheros. Este se construye simplemente combinando listas con metadatos (CommonMetadata y FileMetadata). Se distinguen dos tipos de objetos:

- **Ficheros:** Se envía un FileMetadata.
- **Directorios:** Se envía un CommonMetadata seguido de una lista de objetos (ficheros o directorios).

El parámetro completo consisten en una lista de objetos que representa el directorio raíz (el cuál no tiene metadatos). El árbol se construye recorriendo esta lista y expandiendo los objetos de tipo directorio en esta.

Identificador	1	**Nombre**	OpenShare
Parámetros		**Posibles respuestas**	
1. Propietario (QString) 2. Nombre (QString)		▪ Ok e Id Share (quint64) ▪ ErrorNotExists ▪ ErrorPermissions ▪ ErrorAlreadyOpen	

Tabla 4.4: Comando OpenShare

Notificaciones: Create (3), Delete (4), Change (5) y Rename (6)

Este conjunto de comandos se utilizan para notificar un cambio en los ficheros o directorios supervisados. Pueden enviarse en los dos sentidos (cliente a servidor y viceversa) con semántica similar.

Cuando se envían del servidor al cliente, se utiliza el número de secuencia reservado a las notificaciones e indican un cambio del fichero en otro cliente

Identificador	2	**Nombre**	InitShare
Parámetros		**Posibles respuestas**	
1. Id Share (quint64) 2. Árbol de ficheros		▪ Ok ▪ ErrorNotExists ▪ ErrorAlreadyOpen	

Tabla 4.5: Comando InitShare

(o un cambio local mientras el cliente estaba offline, si se envía durante la sincronización inicial.

El cliente puede enviar estos comandos al servidor para modificar los metadatos almacenados en este, en cuyo caso se considera un comando normal que debe ser respondido. Dado que sólo modifican los metadatos, en caso de haberse modificado los datos, estos deben haberse enviado antes (se almacenarán temporalmente) y este comando completará la operación. Esto facilita la atomicidad de las operaciones, ya que en todo caso, es un sólo comando el que realiza el cambio final.

Los parámetros varían según el tipo de notificación, tal que y como se muestra en las tablas 4.6, 4.7, y 4.8. Los comandos `Create` y `Change` son idénticos con lo que figuran en una sola tabla. En todos los casos, las rutas son absolutas dentro de la compartición, es decir, relativas al punto de montaje.

Identificador	3/5	**Nombre**	Create/Change
Parámetros		**Posibles respuestas**	
1. Ruta padre (QString) 2. Metadatos (Common/FileMetadata)		▪ Ok ▪ ErrorNoData ▪ ErrorPermissions ▪ ErrorNotExists ▪ ErrorAlreadyExists	

Tabla 4.6: Comandos Create y Change

Subidas: StartUpload (7), AbortUpload (8) y UploadComplete (11)

Este grupo de comandos es el utilizado para la señalización de transferencia de ficheros del cliente al servidor. En un momento dado, sólo hay una transferencia de subida en curso, puesto que añadir varias no mejoraría el rendimiento, al estar transmitiéndose toda la información a través de la misma conexión TCP.

Los gestores de descargas pueden aprovechar varias conexiones simultáneas para mejorar el rendimiento, puesto que el sistema operativo dividirá los recursos disponibles entre las distintas conexiones TCP, y usar varias conexiones

Identificador	4	**Nombre**	Delete
Parámetros		**Posibles respuestas**	
1. Ruta (QString)		▪ Ok ▪ ErrorPermissions ▪ ErrorNotExists	

Tabla 4.7: Comando Delete

Identificador	6	**Nombre**	Rename
Parámetros		**Posibles respuestas**	
1. Ruta antigua (QString) 2. Ruta nueva (QString)		▪ Ok ▪ ErrorPermissions ▪ ErrorNotExists ▪ ErrorAlreadyExists	

Tabla 4.8: Comando Rename

proporcionará de más ancho de banda al programa. En este caso, al usarse una sólo conexión (entre otros motivos, porque el objetivo es minimizar el impacto en el resto de actividades del sistema), no se beneficiaría de esta técnica.

Normalmente, se envía `StartUpload` y se espera respuesta para empezar a enviar los datos (a través de otro canal), y cuando se termina el envío, el servidor lo señaliza con `UploadComplete` para indicar que está listo para aceptar otras subidas.

Este comportamiento es similar a "parada y espera" pero a nivel de fichero, y sirve para evitar enviar información identificativa a través del canal de datos. Se podría argumentar que `UploadComplete` es innecesario, puesto que el cliente podría directamente asumir que la descarga se ha terminado al enviar los últimos bytes, enviando a continuación el siguiente comando `StartUpload`. Sin embargo, debido a la caché en la capa de multiplexado, no se garantiza que los bytes del canal de datos se terminaran de enviar antes que el comando, con lo que el servidor rechazaría la subida por haber otra en progreso.

En caso de que el cliente lo desee, puede cancelar la subida en curso con `AbortUpload`, lo que es especialmente útil en caso de que los datos siendo transmitidos hayan dejado de ser válidos, por ejemplo, porque el fichero haya sido modificado durante la transmisión.

Los detalles de `StartUpload` se proporcionan en las tablas 4.9 y 4.10. No se detalla aquí `UploadComplete` puesto que es una notificación y no tiene parámetros ni respuestas.

En caso de recibir `ErrorAlreadyExists` al intentar enviar una transferencia, se entiende que el fichero ya existía en el servidor y no se entiende como un fallo, si no que se interpreta como envío completado con éxito. Esto permite que si un

fichero se crea, borra y recrea no sea necesario enviarlo de menos siempre que el servidor no haya decidido borrar los datos, dependiendo de sus políticas de conservación de copias.

Identificador	7	**Nombre**	StartUpload
Parámetros		**Posibles respuestas**	
1. Id share (quint64) 2. Hash (QByteArray) 3. Tamaño (quint64)		▪ Ok ▪ ErrorBusy ▪ ErrorAlreadyExists	

Tabla 4.9: Comando StartUpload

Identificador	8	**Nombre**	AbortUpload
Parámetros		**Posibles respuestas**	
		▪ Ok ▪ ErrorBusy	

Tabla 4.10: Comando AbortUpload

Descargas: StartDownload (9) y AbortDownload (10)

El funcionamiento de estos parámetros es muy similar a los de subida de datos discutidos en la sección 4.2.2, siendo la principal diferencia la ausencia de un comando análogo a `UploadComplete`. Dado que es en todo caso el cliente el que pide el inicio de las descargas y el que las recibe (pudiendo por tanto detectar su finalización), esta notificación es innecesaria.

El funcionamiento y parámetros de los comandos son similares a los de las descargas, encontrándose los detalles en las tablas 4.11 y 4.12.

Identificador	9	**Nombre**	StartDownload
Parámetros		**Posibles respuestas**	
1. Id share (quint64) 2. Hash (QByteArray)		▪ Ok y Tamaño (quint64) ▪ ErrorBusy ▪ ErrorNotExists	

Tabla 4.11: Comando StartDownload

Identificador	10	**Nombre**	AbortDownload
Parámetros		**Posibles respuestas**	
		▪ Ok ▪ ErrorBusy	

Tabla 4.12: Comando AbortDownload

Mantenimiento de la conexión: Ping (12)

Este comando tiene una doble función: asegurarse de que el otro extremo de la comunicación sigue activo y sincronizar los relojes del cliente y el servidor, tal y como se comenta en la sección Fechas de modificación y relojes del sistema en la página 51. Se aprovecha este comando, puesto que es el único del que se garantiza un envío periódico (necesario para mantener la conexión activa). La hora del servidor se incluye como referencia en las respuestas, como se muestra en la tabla 4.13.

En este sistema, el iniciador de los comandos `Ping` es siempre el cliente, que deberá hacerlo periódicamente para que el servidor no de la conexión por inactiva y la cierre. Inicialmente, el periodo está estipulado en 15 segundos, lo que no debería suponer mucho tráfico de red, mientras que se asegura de que el servidor no desperdicia recursos durante demasiado tiempo.

Identificador	12	**Nombre**	Ping
Parámetros		**Posibles respuestas**	
		▪ Ok y Fecha (QDateTime)	

Tabla 4.13: Comando Ping

Gestión de canales: OpenChannel(13) y CloseChannel(14)

Este par de comandos abren y cierran canales entre el cliente y el servidor. Mediante `OpenChannel` el cliente pide la apertura de un canal de comunicaciones con el resto de clientes conectados a un canal asociado al mismo share y con mismo nombre, siendo el servidor el responsable de asociar un identificador al canal y notificarlo al cliente.

La función contraria se consigue con `CloseChannel`, que dado un identificador de canal, cierra dicho canal de comunicación. Ambos comandos se describen en detalle en las tablas 4.14 y 4.15 respectivamente.

4.3. Implementación de los protocolos

La implementación del protocolo de control forma parte de los módulos de cliente y servidor, puesto que es distinta para cada extremo de la comunicación y fuertemente ligada al funcionamiento de estos módulos. Las únicas clases

Identificador	13	**Nombre**	OpenChannel
Parámetros		**Posibles respuestas**	
1. Id share (quint64) 2. Nombre canal (QString) 3. Tipo (bool)		▪ Ok e Id canal (quint16) ▪ ErrorBusy	

Tabla 4.14: Comando OpenChannel

Identificador	14	**Nombre**	CloseChannel
Parámetros		**Posibles respuestas**	
1. Id canal (quint16)		▪ Ok ▪ ErrorNotExists	

Tabla 4.15: Comando CloseChannel

comunes para este protocolo son `CommonMetadata` y `FileMetadata`, simples contenedores de información que implementan métodos para serialización. También se dispone de una clase de utilidad: `Serializer` que simplifica el funcionamiento de las clases de Qt (`QDataStream`), reiniciando el buffer a su posición inicial en caso de un error de lectura (normalmente porque el mensaje no ha sido recibido completamente en el momento de interpretarlo).

Debido al diseño en varios niveles, el protocolo de multiplexación es independiente de la funcionalidad, lo que permite su implementación en código común y reutilizable. La implementación del protocolo se divide en tres clases como se muestra en la figura 4.4, heredando todas de las clases correspondientes de Qt.

4.3.1. SocketChannel

`SocketChannel` representa cada uno de los canales de comunicación abiertos. Para facilitar su uso se implementa como un dispositivo de entrada/salida o `QIODevice` que proporciona funciones de lectura y escritura que aceptan diversos parámetros con sólo implementar las primitivas `readData()` y `writeData()`.

Para enviar datos almacena los datos en un buffer y notifica a través del mecanismo de señales de Qt al `MultiplexerSocket` asociado, para que los transmita. También utiliza este mecanismo para notificar a la clase usuario de la disponibilidad de espacio en el buffer de salida para escritura.

El funcionamiento de la recepción de datos es muy similar, con la diferencia de ser `MultiplexerSocket` el que escribe en el buffer y la clase usuario la que los lee.

4.3.2. MultiplexerSocket

`MultiplexerSocket` implementa el protocolo y las políticas de prioridad. Deriva de la clase `QSslSocket` que implementa todos los detalles de TLS y SSL,

que a su vez deriva de `QTcpServer` que implementa la gestión de la conexión TCP.

Mantiene una lista de canales (objetos `SocketChannel`) que componen la interfaz que se le ofrece al desarrollador de la aplicación. Cuando se reciben datos, se procesa la cabecera y se escriben los datos al canal correspondiente, manteniendo el estado en el que se detuvo la lectura, es decir: el canal al que se estaba escribiendo y el número de bytes restantes antes de esperar otra cabecera.

Cada vez que se recibe una señal de disponibilidad de datos de un canal, se procede a enviar los datos de dicho canal siempre que no haya un envío en curso en ese momento. En ese caso se envía la petición hasta que se termine el envío actual, momento en el cuál se procederá a recorrer la lista de canales en el orden indicado por las prioridades (ver subsección 4.1.2) y se procederá a enviar todos los datos disponibles en el canal de mayor prioridad, repitiendo el proceso hasta vaciar todos los buffers de envío de los canales.

4.3.3. MultiplexerSocketServer

`MultiplexerSocketServer` se utiliza sólo en el servidor para esperar y aceptar conexiones de clientes. Se utiliza una clase derivada de `QTcpServer` para abrir las nuevas conexiones como instancias de `MultiplexerSocket` en vez del comportamiento por defecto de crear objetos de tipo `QTcpSocket`.

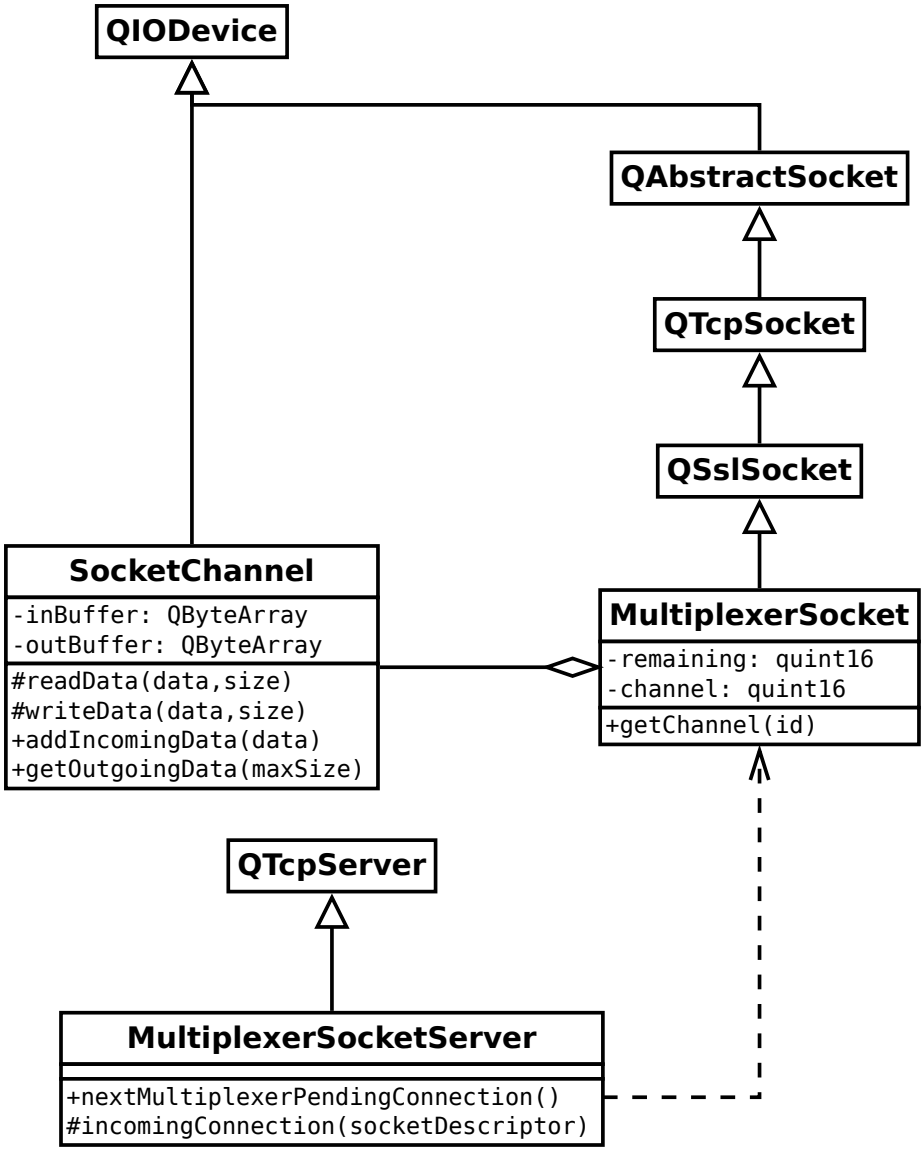


Figura 4.4: Diagrama de clases simplificado para el protocolo de multiplexado

Capítulo 5

Diseño del cliente

El cliente es la parte del sistema ejecutada en los equipos a sincronizar y se ocupa de propagar los cambios locales a través de la red y efectuar localmente los cambios realizados en otros equipos. Para ello, dispone de dos componentes fundamentales: la conexión al servidor y la operativa sobre el sistema de ficheros local. Debe ser capaz de conectarse a varios servidores y a varios shares dentro del mismo servidor, con sus correspondientes puntos de montaje locales. Finalmente, debe proporcionar información al usuario sobre el estado del sistema en caso de que lo desee, manteniéndose oculto en la medida de lo posible en caso contrario.

Durante el diseño, hay que recordar los objetivos fundamentales del sistema, por lo que habrá que localizar los posibles puntos de inserción de plugins y las capacidades que se les asignará a estos, de forma que se permita la máxima personalización sin que puedan afectar al funcionamiento básico del sistema, o al menos minimizar este efecto.

Otro punto a considerar es el rendimiento lo que, en el caso de procesadores multinúcleo modernos, implica la paralelización de tareas. En este aspecto, se identificarán las tareas más costosas y se intentarán dividir de forma que puedan ser ejecutadas por varios hilos para aprovechar al máximo las capacidades del equipo, sin olvidar la posibilidad de que el usuario prefiera una ejecución menos paralela para que el programa no afecte al rendimiento de otras aplicaciones que puedan estar ejecutándose simultáneamente.

5.1. Diseño modular

La división del programa en módulos viene dada por cada uno de los niveles a los que puede pertenecer una funcionalidad. Ya que una misma instancia del cliente permitirá conectar a varios servidores y utilizar varios puntos de montaje, se debe separar la lógica del programa entre la perteneciente a una conexión con el servidor, y la perteneciente a cada uno de los puntos de montaje. Es más, esto último podría dividirse en dos partes fundamentales, la encargada de conversar con el servidor y la responsable de las operaciones sobre el sistema de ficheros.

Así, obtenemos los siguientes bloques en el cliente, ordenadas de menor a mayor número de instancias de cada uno, es decir, de funcionalidad global al detalle:

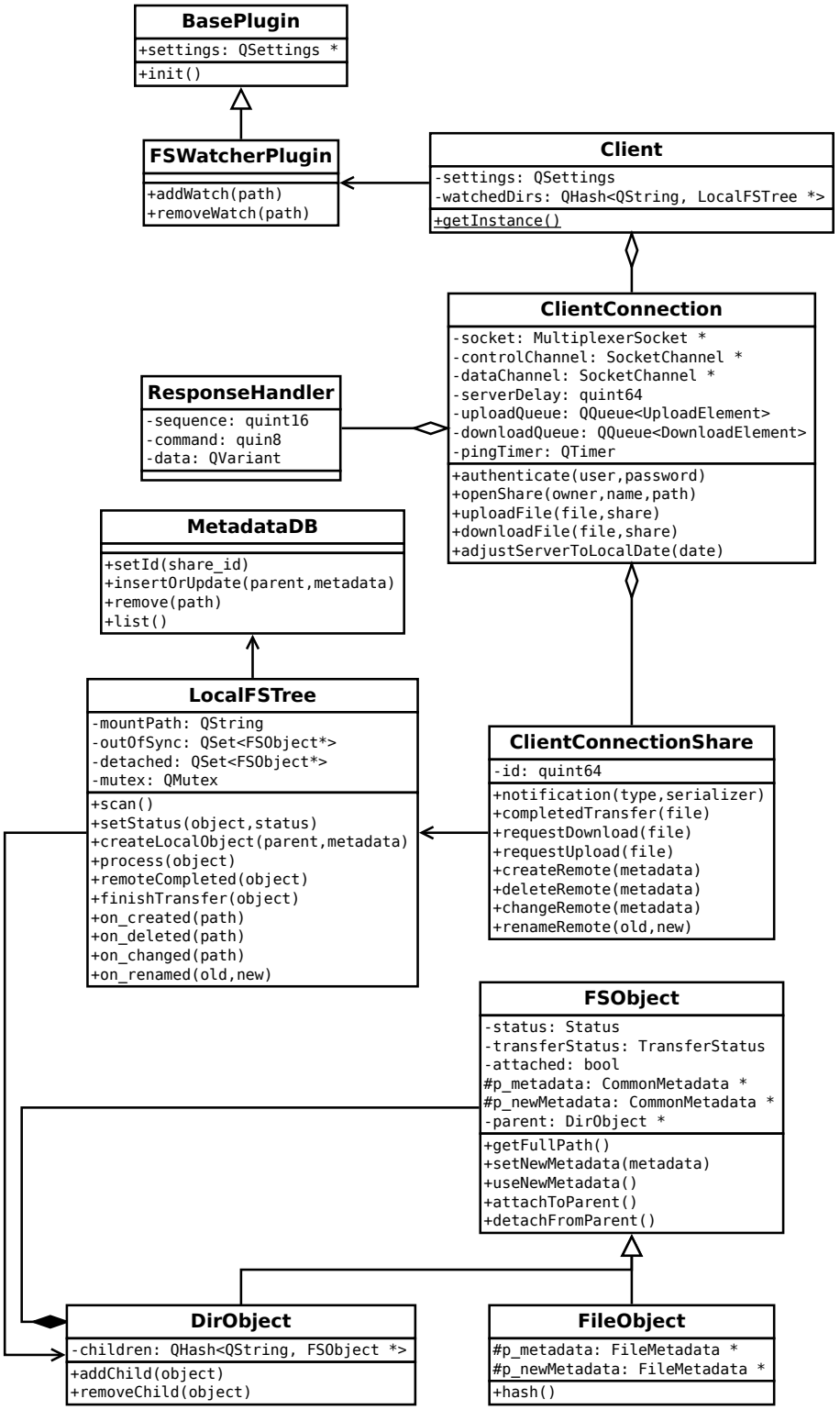


Figura 5.1: Diagrama de clases simplificado para el cliente

1. **Cliente:** Contiene la funcionalidad común al cliente como la gestión configuración o la carga de plugins. Una sola instancia.

2. **Conexión:** Una conexión con el servidor y gestiona la comunicación no específica a los shares como el login. Se crea una instancia por cada conexión a un servidor activa.

3. **Share:** La mayor parte de la funcionalidad del cliente se engloba en este módulo, responsable de la sincronización de los ficheros de un punto de montaje con la compartición correspondiente del servidor. Es necesaria una instancia por cada compartición abierta. Puede dividirse en dos submódulos:

   - **Conexión a Share:** Submódulo encargado del envío y recepción de los mensajes de red relativos a un share.
   - **Gestión del sistema ficheros:** Todas las operaciones sobre el sistema de ficheros pasan a través de este módulo, así como las notificaciones de cambios del sistema de ficheros. También es responsable de mantener un listado de las diferencias con el servidor y resolverlas,

Algunos de estos módulos se divide en varias clases, mientras que otros pueden implementarse en una única clase como se muestra en el diagrama de la figura 5.1. Por brevedad, se han omitido clases de apoyo y gran cantidad de métodos auxiliares que no son relevantes para la comprensión general del funcionamiento del cliente. Todas estas clases, junto con sus principales métodos y atributos se discutirán en las siguientes secciones.

5.2. Representación del árbol local: FSObject

Una de las funciones más importante del cliente es mantener actualizada la información sobre el sistema de ficheros local, así como los cambios necesarios para mantener esta sincronización. Esta funcionalidad se implementa en la clase `LocalFSTree`, apoyándose en `FSObject` y sus clases derivadas como forma de representar los datos. Para ello, contiene una referencia a un objeto `DirObject` que representa el directorio local que está siendo sincronizado.

Todo `DirObject` se compone de una lista de objetos hijo, ya sean ficheros o directorios. Estos se almacenan en una estructura de datos tipo mapa, en la que se asocia el nombre del objeto (clave) con la referencia al objeto que contiene toda la información (valor), lo que permite una navegación por nombre muy rápida.

Además, al ser una clase derivada de `FSObject` contiene todos los atributos de esta, a saber:

- Referencia al objeto de directorio padre (`parent`) o `NULL` para el raíz.
- Metadatos del objeto (`p_metadata` del tipo CommonMetadata o FileMetadata, descritos en la subsección 4.2.1).
- Nuevos metadatos del objeto(`p_newMetadata`). Utilizados como almacenamiento temporal durante las operaciones de cambio o creación local.

- Estado de sincronización (`status`) y de transferencia(`transferStatus`, descrito a continuación.
- Si este objeto está unido al árbol principal (`attached`).

Los posibles estados de sincronización son los mostrados en el listado de código 5.1, consistentes en las 8 posibilidades de combinación de las 4 operaciones del sistema (crear, modificar, borrar y renombrar) con los dos puntos sincronizados (cliente y servidor), además del estado sincronizado. En el listado de código 5.2 se muestran los estados de transferencia de un fichero, existiendo dos estados para la operación en curso, dependiendo de si está se encuentra encolada o no y otro para el estado inicial o reposo.

```
enum Status {
        NeedsRemoteCreation, //Existe localmente,
            necesita crearse en el servidor (subir)
        NeedsRemoteDeletion, //No existe localmente,
            necesita borrarse en el servidor
        NeedsRemoteChange, //Es necesario propagar la
            versión local al servidor (subir)
        NeedsRemoteRename, //Es necesario renombrar la
            versión del servidor
        NeedsLocalCreation, //Existe en el servidor,
            necesita crearse localmente (descargar)
        NeedsLocalDeletion, //No existe en el servidor,
            es necesario borrar la copia local
        NeedsLocalChange, //Es necesario obtener la versi
            ón del servidor (descargar)
        NeedsLocalRename, //Hay que renombrar la versión
            local para que coincida con el servidor
        InSync //Sincronizado
};
```

Listado de código 5.1: Estados de sincronización

```
enum TransferStatus {
        NotStarted, //Transferencia necesaria y pendiente
        Queued, //Transferencia en cola o en proceso
        Done //Transferencia completada o innecesaria
};
```

Listado de código 5.2: Estados de transferencia

Estas clases implementan pocos métodos aparte de los necesarios para el acceso a sus atributos (getters y setters). El par de métodos `setNewMetadata()` y `useNewMetadata()` se utilizan para utilizar nuevos metadatos cuando se produce un cambio en el fichero, pero en dos fases: primero se llama a `setNewMetadata` en cuanto los nuevos metadatos están disponibles, y cuando el objeto ha sido

sincronizado se ejecuta `useNewMetadata()` que moverá los nuevos metadatos a `p_metaData` para su acceso normal. Esto permite almacenar metadatos temporales mientras dura una operación de sincronización, de forma que los metadatos referenciados por `p_metaData` se refieran siempre al estado actual de sincronización.

Todos los objetos disponen además de `getFullPath()` que devuelve la ruta completa del objeto, construida recorriendo sus padres hasta llegar a la raíz. También relacionados con la gestión del árbol existen `addChild()` y su compañero `removeChild()` en `DirObject`, los cuáles añaden y borran objetos al mapa de hijos del directorio, además de ajustar el puntero padre del hijo.

Para gestionar la separación y unión de objetos al árbol principal de directorios, se utilizan los métodos `attachToParent()` y `detachFromParent()`, que modifican el valor del atributo `attached`. Además, al separarse se modifican algunos detalles más como grabar la ruta completa al objeto (puesto que al separarse de su padre no podrá recuperarse con la función `getFullPath()`) y borrarse de la lista de hijos de su padre. Estas modificaciones se deshacen al volver a llamar a `attachToParent()`.

Finalmente, los objetos que representan ficheros tienen el método `hash()` que ejecuta el algoritmo de hash seleccionado (actualmente SHA-1 tal y como se discute en la sección 9.1) sobre los datos del fichero. Se tiene en cuenta la fecha de modificación para no rehashear ficheros no modificados, ya que suele ser una operación lenta. Además, es una operación fácilmente paralelizable, lo que se consigue a través de una lista de tareas y un grupo de hilos, todo gestionado automáticamente por Qt con sus clases `QRunnable` (interfaz implementada por una clase de utilidad no mostrada en el diagrama de clases para hashear ficheros) y `QThreadPool` que manejar el conjunto de hilos automáticamente, ajustando el número de hilos en función de las características del sistema.

5.3. Gestión del árbol local: LocalFSTree

La mayor parte de la lógica de sincronización se encuentra en esta clase, que dispone de las siguientes formas de obtener información sobre el estado local y remoto del directorio sincronizado:

- El método `scan()`, que escanea el directorio y construye el árbol de objetos inicial.
- Los slots para cada evento (`on_created()`, `on_changed()`...) que reciben información en tiempo real de cambios en el sistema de ficheros local, generada por señales del plugin de monitorización de directorios (implementa la interfaz `FSWatcherPlugin`).
- Los slots relativos a la red: `remoteCompleted()`, `finishTransfer()` y otros para informar de fallos, conectados a las señales correspondientes de la clase `ClientConnectionShare`.

Con toda la información mantiene un árbol de objetos (`FSObject`) y su estado, que siempre se refiere al estado del sistema de ficheros local, de forma que si se recibe una notificación del servidor, no se actualizará el árbol hasta que se hayan realizado los cambios correspondientes al sistema de ficheros.

Además de este árbol, se mantiene por conveniencia un conjunto con referencias a los objetos no sincronizados en el atributo `outOfSync`, para poder acceder rápidamente a los objetos que necesitan procesado. Igualmente se mantiene en `detached` el conjunto de objetos separados del árbol principal, una funcionalidad necesaria para representar objetos en creación que todavía no existen en el sistema de ficheros local y por tanto no tienen lugar en dicho árbol.

La gestión de estas listas se efectúa únicamente en los métodos `setStatus()` y `createLocalObject()`, de forma que se garantiza que los datos en dichas listas y en el árbol son consistentes. Además, puesto que cada árbol de ficheros se ejecuta en su propio hilo para paralelizar las operaciones de acceso a disco, se utiliza un mutex para controlar el acceso a las listas, ya que ambos métodos pueden ser llamados desde el hilo de ejecución principal, también ocupado de las comunicaciones en red.

Otra gran ventaja de esta exclusividad de acceso a los estados de los objetos, es que puede utilizarse el método `setStatus()` para unir notificaciones sobre el mismo fichero, puesto que puede acceder al estado antiguo y el nuevo sin problemas. Así por ejemplo, si un fichero está en el estado de `NeedsLocalCreation` y se pretende pasar al estado `NeedsLocalChange`, se pueden unir ambas notificaciones manteniendo el estado y cambiando los metadatos del fichero a los adjuntos al último cambio de estado, abortando la descarga en curso en caso de que se tratara de un fichero y los nuevos metadatos indiquen un hash distinto al de la descarga en curso. El comportamiento se puede analizar en detalle en el listado de código 5.3, dónde se omite el código de los cambios de nombres, puesto que estos se resuelven inmediatamente, o se dividen en dos tareas: borrado del fichero antiguo y creación del fichero nuevo, con la ventaja de poder reutilizar el código para estas acciones.

Finalmente, el método `process()` es llamado para todo objeto no sincronizado y ejecutará los métodos auxiliares necesarios para que pase a estarlo, incluyendo la petición de descargas o subidas a través de `ClientConnectionShare`, la creación y borrado de ficheros (recursivamente en caso de tratarse del borrado de un directorio), etc.

La ventaja de utilizar un único punto de entrada a estas modificaciones es que hace más fácil gestionar los reintentos en caso de fallo temporal como puede ser un fichero bloqueado o falta de permisos. Al mantener una lista con los ficheros no sincronizados, es tan sencillo como llamar periódicamente al método `process()` sobre cada elemento de ese conjunto, lo cuál puede resolverse con un único temporizador.

Finalmente, durante todo este proceso se mantiene actualizada una base de datos con los metadatos de todos los ficheros sincronizados, con el objetivo de acelerar el escaneo del directorio sincronizado siempre que no haya sido modificado. El formato y funcionamiento de esta base de datos se detalle en la sección 5.4.

5.4. Base de datos de metadatos: MetadataDB

La base de datos de metadatos es utilizada por el cliente para guardar información sobre los metadatos más relevantes de los objetos sincronizados. Esto sirve para acelerar el arranque del sistema, dado que si el objeto real en el sistema de ficheros tiene una fecha de modificación coincidente con la de la base

```
switch (oldStatus) {
case FSObject::NeedsLocalChange: //No break
case FSObject::NeedsLocalCreation: {
        if (FSObject::statusLocal(st)) {
                //Otro cambio remoto, mantener
                if (file && o->getTransferStatus() != FSObject::Done) {
                        [...] //Abortar descarga si es necesario
                }
                if (st == FSObject::NeedsLocalDeletion) {
                        [...] //Borrado remoto, abortar creación
                }
        } else {
                //Cambio local, enviar cambio al servidor
                if (file && o->getTransferStatus() != FSObject::Done) {
                        share->abortDownload(file);
                }
                o->setStatus(st);
        }
} break;

case FSObject::NeedsLocalDeletion: {
        //Fichero pendiente de borrado local
        if (st == FSObject::NeedsRemoteDeletion) {
                [...] //Fichero borrado en local, completado
        } else {
                //Cambiar al nuevo estado
                o->setStatus(st);
        }

} break;

case FSObject::NeedsRemoteChange: //No break
case FSObject::NeedsRemoteCreation: {
        if (FSObject::statusLocal(st)) {
                //Cambio en el servidor, pasar a descargar
                if (file && o->getTransferStatus() != FSObject::Done) {
                        [...] //Abortar descarga si es necesario
                }
                o->setStatus(st);
        } else {
                //Más cambios locales, unir si es posible
                if (!o->getRequestInFlight()) {
                        //Transferencia en progreso/en cola, reiniciar
                            subida
                        if (st == FSObject::NeedsRemoteDeletion &&
                            oldStatus == FSObject::NeedsRemoteCreation)
                             {
                                [...] //Cancelar creación
                        } else {
                                [...] //Reiniciar subida
                        }
                } else {
                        //Operación terminando, completar proceder con
                            el siguiente cambio
                        o->requestFinished();
                        o->setStatus(st);
                }
}
} break;

case FSObject::NeedsRemoteDeletion: {
        if (FSObject::statusLocal(st)) {
                //Ignorar eventos del servidor
        } else {
                //Si se recrea localmente, cambiar a actualización
                o->requestFinished();
                o->setStatus(st);
        }
} break;
}
```

Listado de código 5.3: Código resumido de la unión de cambios de setStatus()

de datos, no es necesario re-escanear dicho objeto. En caso de los directorios, la ganancia es mínima, pero en los ficheros se puede ahorra calcular el hash de sus datos, lo que representa importantes ganancias en consumo de CPU y acceso a disco.

Además, se almacena el código de versión del servidor, de manera que pueda enviarse con el comando `InitShare`, con lo que se evita comparar todos los metadatos para comprobar si los objetos coinciden puesto que es suficiente comparar el número de versión.

Toda esta información se almacena en una única tabla (`objects`) definida con la sentencia mostrada en el listado de código 5.4, compuesta por los siguientes campos:

- `share_id`: El identificador del share al que pertenece el directorio. Sirve para diferenciar objetos con la misma ruta relativa pero en distintos puntos de montaje.
- `path`: La ruta dentro de la compartición e identificador principal del objeto.
- `version`: Identificador de versión proporcionado por el servidor o `NULL` si se desconoce, por ejemplo porque el objeto haya sido modificado.
- `lastModified`: Fecha de última modificación.
- `hash`: Hash de los datos del fichero o `NULL` para directorios.

```
CREATE TABLE objects (
        share_id UNSIGNED BIG INT,
        path TEXT,
        version UNSIGNED BIG INT,
        lastModified DATETIME,
        hash BLOB,
        PRIMARY KEY(share_id, path)
)
```

Listado de código 5.4: Definición de la base de datos de metadatos

El motor de base de datos utilizado es SQLite [23] por no necesitar de ningún tipo de servidor externo, integrándose completamente en la aplicación, además de tener excelente soporte en Qt, a través del driver QSQLite. Además, esto puede facilitar la portabilidad a dispositivos móviles puesto que muchos de ellos (incluyendo los que utilizan los sistemas operativos iOS y Android) soportan este tipo de bases de datos de forma nativa.

El acceso a la base de datos se realiza a través de la clase `MetadataDB`, que se ocupa de convertir los objetos del formato del programa al de la base de datos, proporcionando métodos para el acceso, inserción, modificación y borrado de los elementos almacenados. Cabe destacar que las únicas lecturas a la base de datos se realizan durante la carga inicial del programa, durante la que se lee toda la información, por tanto no hay un método para ejecutar consultas personalizadas.

5.5. Comunicación con el servidor: ClientConnection y ClientConnectionShare

Este par de clases implementan la comunicación con el servidor. Todos los mensajes, tanto enviados como recibidos, pasan a través de `ClientConnection` que es la ocupada de gestionar el socket para la conexión al servidor, por muchas comparticiones que estén abiertas en este. También gestiona las descargas y subidas, puesto que sólo puede haber una transferencia en cada dirección activa por servidor por las razones discutidas en la sección 4.2.2.

Por otro lado, `ClientConnectionShare`, recibe y origina todos los mensajes relacionados con una compartición, además de gestionar la toma de decisiones sobre la resolución de conflictos.

5.5.1. ClientConnection

Una conexión al servidor se crea durante el arranque del programa, además de cada vez que es interrumpe la conexión al servidor y es necesario reconectar. Inicialmente, el objeto de tipo `Client` llama al método `authenticate()` con las credenciales proporcionadas por el usuarios y efectúa una llamada a `openShare()` por cada compartición a las que el usuario quiera acceder en este servidor.

Inicio de conexión

Durante todo este proceso, se crea un objeto de `ClientConnectionShare` por cada compartición abierta y se envían los correspondientes mensajes al servidor, en primer lugar `Login` y a continuación múltiples `OpenShare`. Si todo va bien, el control de las conexiones pasa a los objetos ocupados de cada una de ellas, quedando como únicas responsabilidades de esta clase el mantenimiento de la conexión (comando `Ping`) y las transferencias de ficheros. En caso de error, se comunica a la clase principal del cliente a través de una señal de Qt, `error`, incluyendo un mensaje de error para mostrar al usuario.

Gestión de respuestas

Además, puesto que todos los comandos pasan a través de esta clase, se utiliza para gestionar las acciones ante las respuestas del servidor, utilizando la clase auxiliar `ResponseHandler`, utilizada para almacenar información sobre todos los mensajes enviados: el número de secuencia, el tipo de comando y un campo libre dependiente del comando enviado. Por cada comando enviado, se almacena uno de estos objetos en una lista ordenada por número de secuencia.

De esta forma, al recibir una respuesta, puede localizarse su número de secuencia en dicha lista y ejecutar la acción adecuada dependiendo del comando. Muchas de estas acciones serán simplemente pasar el mensaje a la clase que representa la conexión con la compartición adecuada, utilizando el campo de datos libres para almacenar el identificador de esta, mientras que otros (por ejemplo, los relacionados con el login o las transferencias) se gestionarán internamente.

Mantenimiento de conexión

Respecto a mantener activa la conexión, se envía periódicamente el comando `Ping` utilizando un temporizador, emitiendo un error en caso de no haber recibido respuesta en el momento de tener que enviar el siguiente. La hora del servidor recibida en cada respuesta a estos comandos se almacena en forma de diferencia horaria con la hora del sistema y se proporciona el método `adjustServerToLocalDate()` para convertir horas locales a horas del servidor para su envío y comparación.

Transferencias

Finalmente, las transferencias se gestionan utilizando una cola para cada sentido, es decir, subidas y descargas. Siempre que haya algún elemento en estas colas y no se esté efectuando una transferencia en esa diversión, se extraerá el primer elemento de la cola y se iniciará la transferencia. Cuando la transferencia finalice o sea abortada por algún error, se notificará a la compartición que solicitó esta transferencia a través de los métodos `uploadFile()` o `downloadFile()`.

En las colas se almacena la siguiente información sobre cada elemento:

- Share peticionaria
- Ruta del fichero
- Referencia al `FileObject` correspondiente
- Tamaño y hash de los datos

Durante las descargas, se almacenan los datos descargados en un fichero temporal localizado en un directorio de ficheros temporales dependiente del sistema operativo subyacente (por ejemplo /tmp en la mayoría de los sistemas operativos tipo Unix). De esta forma, el fichero original no es modificado hasta que se ha completado la descarga con éxito, lo que se verifica utilizando el hash de los datos del fichero. Por razones de rendimiento, el hash se va calculando cada vez que se reciben datos del servidor, de forma que no es necesario recorrer el fichero completo una vez descargado. La responsabilidad de copiar el fichero temporal a su localización final es de `LocalFSTree`, para centralizar todas las operaciones de disco en una sólo lugar, además de ejecutarse en su hilo propio.

Para las subidas, se empieza a transferir el fichero directamente desde su localización original, para evitar una copia innecesaria a una localización temporal. Sin embargo, en caso de que el fichero sea modificado durante la transferencia, podrían enviarse datos corruptos al servidor. Para evitar esto se dispone de dos contramedidas: el monitoreo del sistema de ficheros que aborta la transferencia si los datos cambian y un hash calculado sobre los datos enviados que en caso de detectar un error al enviar los últimos datos, enviará en su lugar el comando `AbortUpload` para cancelar la transferencia.

5.5.2. ClientConnectionShare

Esta clase actúa como un intermediario entre la conexión que reside en `ClientConnection` y el árbol local representado por `LocalFSTree`, recibiendo los mensajes referidos a la compartición del primero informando al segundo

y recibiendo peticiones del árbol local para construir el mensaje a mandar a través de la conexión.

Los métodos `notification()` y `completedTransfer()` reciben las notificaciones del servidor. Para las transferencias simplemente se informa al árbol de que ha finalizado (correcta o incorrectamente) para que pueda actuar acordemente. Sin embargo, para las notificaciones de cambios, es esta clase la encargada de asignar el nuevo estado de sincronización (llamando a `setStatus()`) al objeto, dependiendo del estado actual de dicho objeto y los metadatos asociados.

Por defecto, se implementa la política de resolución de conflictos detallada en la subsección 3.6.1, para lo que se dispone de acceso a los metadatos antiguos y nuevos, lo que incluye la versión del objeto y la fecha de última modificación.

Por otra parte, los métodos `*Remote()` crean un mensaje para realizar la acción correspondiente sobre el objeto en el servidor, incluyendo los parámetros necesarios, y la envían a través de la conexión correspondiente. La respuesta a este comando también se interpretará en esta clase, notificando al árbol local del éxito o fracaso de la operación.

5.6. Funciones globales: Client

En el singleton `Client` residen todas las funciones globales del cliente, incluyendo la gestión de la configuración, la carga dinámica de plugins, la gestión de conexiones y la comunicación con el usuario.

5.6.1. Configuración

Para la gestión de la configuración se ha decidido utilizar la clase `QSettings` incluida en Qt, puesto que esta almacenará los datos de configuración en una ubicación adecuada al sistema operativo subyacente de forma automática, aunque en todo caso puede utilizarse el sistema de ficheros de configuración.

En el listado de código 5.6 se muestra el formato genérico soportado en todos los sistemas operativos y utilizado por defecto en Linux (dónde se almacenará en un fichero en el directorio `.config` del home del usuario). En Windows la configuración se almacena por defecto en el registro del sistema como se muestra en la figura 5.2. Por último, en Mac OS X se almacena en ficheros `.plist` que pueden ser almacenados en formato binario y XML, mostrándose este último en el listado de código 5.5.

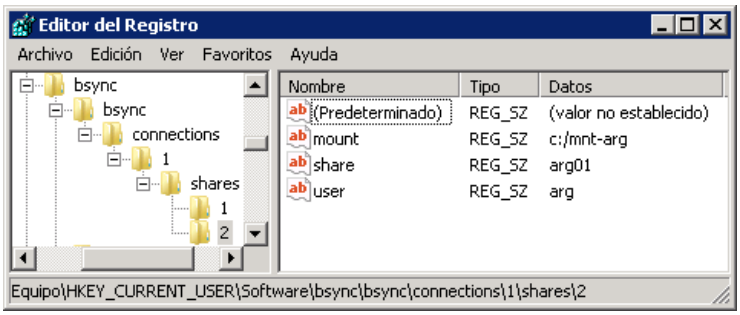


Figura 5.2: Ejemplo de configuración en el registro de Windows

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
 "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>connections</key>
  <array>
    <dict>
      <!-- Configuración de la conexión 1 -->
      <key>address</key>
      <string>nadir.uc3m.es</string>
      <key>user</key>
      <string>jtn</string>
      <key>password</key>
      <string>jtn.bsync</string>
      <key>shares</key>
      <array>
        <dict>
          <!-- Configuración del share 1 -->
          <key>share</key>
          <string>jtn01</string>
          <key>mount</key>
          <string>/tmp/mnt-jtn</string>
        </dict>
        <dict>
          <!-- Configuración del share 2 -->
          <key>user</key>
          <string>arg</string>
          <key>share</key>
          <string>arg01</string>
          <key>mount</key>
          <string>/tmp/mnt-arg</string>
        </dict>
      </array>
    </dict>
  </array>
</dict>
</plist>
```

Listado de código 5.5: Ejemplo de configuración en formato de plists XML

```
[connections]
; Número de servidores
size=1

; Configuración de la conexión 1
1\address=nadir.uc3m.es
1\user=jtn
1\password=jtn.bsync

; Número de shares del servidor 1
1\shares\size=2

; Configuración del share 1
1\shares\1\share=jtn01
1\shares\1\mount=/tmp/mnt-jtn

; Configuración del share 2
1\shares\2\user=arg
1\shares\2\share=arg01
1\shares\2\mount=/tmp/mnt-arg
```

Listado de código 5.6: Ejemplo de configuración mínima en fichero de texto

Este sistema permite el almacenamiento de pares de clave-valor, con claves anidadas y arrays de valores. La configuración más relevante del cliente se almacena en el array de nombre `connections`, en los que cada entrada contiene los valores necesarios para conectarse a un servidor:

- Dirección del servidor, ya sea un nombre de dominio o dirección IP.
- Puerto TCP del servidor, por defecto 48135.
- Usuario y contraseña. Consultar las consideraciones de seguridad en la subsección 9.2.1.
- Un array conteniendo la información de cada compartición a abrir:
  - Propietario de al compartición. Por defecto el mismo usuario de login.
  - Nombre de la compartición.
  - Ruta del punto de montaje local, usando siempre la barra hacia adelante (/) como separador de ruta.

Se pueden sobrescribir también los directorios de ficheros temporales, la localización de la base de datos o el tamaño máximo de la caché de ficheros descargados, pero la otra función principal de esta configuración es almacenar los nombres de los plugins a utilizar. Todo esto se guarda bajo varias claves dentro de la superclave `plugins`. Por ejemplo, la clave `plugins/watcher` contiene el nombre del plugin de monitoreo del sistema de ficheros a utilizar. Una clave adicional, `plugins/dir` almacena el directorio donde se sitúan los plugins en caso de ser distinto del directorio por defecto.

5.6.2. Carga de plugins

Mediante el uso de `QPluginLoader` se cargan todos los plugins listados en la configuración y se crean las instancias necesarias de las clases que implementan los plugins. Los plugins se buscan por defecto en el directorio `plugins` de la aplicación, considerando todos los ficheros disponibles. Esto puede producir advertencias durante el arranque si ciertos ficheros no son librerías dinámicas, pero evita la necesidad de utilizar extensiones al nombre de fichero, detectando los plugins por sus contenidos.

El único plugin obligatorio es el de monitoreo de sistemas de ficheros (ver detalles de la interfaz del plugin y su implementación en la sección 7.2), de manera que si no hay ninguna entrada en el fichero de configuración se buscarán todos los plugins de este tipo y se mostrará un error en caso de haber más de uno o ninguno, cargando el único disponible en otro caso. Además, únicamente se crea una instancia de este tipo de plugin, por lo que la gestión de adición y borrado de directorios a monitorizar se realiza en esta clase mediante los métodos `addWatch()` y `removeWatch()`, que serán llamados desde el árbol local durante su creación y borrado.

5.6.3. Interfaz con el usuario

Todos los errores mencionados se imprimen por pantalla por defecto mediante el uso del sistema incluido en Qt, utilizando la clase `QDebug` que permite enviar mensajes de varias prioridades y gestionarlos a través de una función personalizada que puede configurarse llamando a `qInstallMsgHandler()`. Por defecto, se utiliza la salida de error (stderr) para mostrar los errores, y queda a cargo del plugin de interfaz utilizar otro sistema.

5.6.4. Gestión de conexiones

Finalmente, esta clase gestiona las conexiones al servidor, abriendo todas las conexiones configuradas durante el arranque. Si en algún momento alguna conexión se pierde, emitirá la señal correspondiente y el cliente cerrará la conexión, borrando adecuadamente los recursos usados. La conexión se reintentará al cabo de unos segundos, utilizando una técnica de "exponential back-off" para no saturar al servidor.

Capítulo 6

Diseño del servidor

Dentro del sistema completo, el servidor es la parte encargada de sincronizarse con los clientes, de forma muy similar a lo que hacen estos. Sin embargo, hay dos diferencias fundamentales entre cliente y servidor: un servidor debe aceptar conexiones de varios clientes y facilitar al comunicación entre ellos (siempre a través del servidor) y proporcionar funcionalidades adicionales no disponibles en el cliente, como es el archivado de versiones antiguas de los objetos sincronizados.

Es precisamente para poder proporcionar el archivado de documentos que se decide utilizar una base de datos como backend del servidor, en vez de utilizar directamente el sistema de ficheros como el cliente. De esta forma se pueden archivar fácilmente los metadatos de versiones antiguas.

Sin embargo, no es conveniente guardar los datos de los ficheros dentro de la base de datos puesto que su gran tamaño ralentizaría el funcionamiento global de esta. Por ello, se recurre al sistema de ficheros como backend para los datos, para lo que necesitamos una forma única de identificar los datos.

Una posibilidad es utilizar como identificador el nombre del fichero y su número de versión combinados adecuadamente, lo que garantiza la unicidad de los nombres de ficheros para todos los datos almacenados. Sin embargo, dado que el sistema ya utiliza hashes para detectar diferencias entre ficheros, podemos utilizar estos como nombre del fichero de datos. Finalmente, otra opción es utilizar un nombre aleatorio e indicarlo en la base de datos.

El prototipo implementa la segunda opción, utilizar los hashes de los datos como nombre de ficheros, lo que tiene la ventaja de que en caso de que dos ficheros tengan los mismos datos (por ejemplo, por ser uno copia de otro) no ocuparán el doble de espacio, además de no requerir información adicional en la base de datos y ser trivial verificar la integridad de los ficheros almacenados. Las consideraciones de seguridad de esta elección se describen en detalle en la sección 9.1.

6.1. Diseño modular

El diseño del servidor es similar al cliente, por tener que tratar problemas similares. Por tanto, tal y como ya se explicó en la sección 5.1 se necesita un módulo para gestionar la funcionalidad global, otro para cada conexión y otro

para cada compartición. En el servidor se necesita además una conexión con la base de datos que sustituye al sistema de ficheros en el cliente.

Así, obtenemos los siguientes bloques en el cliente, ordenadas de menor a mayor número de instancias de cada uno, es decir, de funcionalidad global al detalle:

1. **Servidor:** Contiene la funcionalidad común al cliente como la gestión configuración o la carga de plugins. Una sola instancia.

2. **Conexión:** Cada conexión con los clientes, gestiona la comunicación no específica a los shares. Se crea una instancia por cada conexión a un servidor activa.

3. **Share:** Engloba el procesado de todos los mensajes relativos a shares así como la notificación de cambios a los clientes. Es necesaria una instancia por cada compartición abierta. Puede dividirse en dos submódulos:

   - **Conexión a Share:** Submódulo encargado del envío y recepción de los mensajes de red relativos a un share.
   - **Modelo de base de datos:** Una abstracción sobre la base de datos para permitir su uso por múltiples conexiones, ya que varios clientes pueden conectarse al mismo share.

Estos módulos son implementados por las clases mostradas en la figura 6.2, donde se han omitido clases de apoyo y gran cantidad de métodos auxiliares que no son relevantes para la comprensión general del funcionamiento del servidor. A continuación, se detallará el funcionamiento de cada una de estas clases.

6.2. Base de datos

El diseño de la base de datos está replicado tanto en el motor de base de datos como en la capa de acceso a datos dentro del programa del servidor. En esta sección se va a describir la base de datos desde el punto de vista del sistema gestor, mencionando brevemente las características de la capa de acceso a datos, puesto que esta no es más que una abstracción para evitar tener que manipular código SQL cada vez que se requiera acceder a datos.

A continuación se describen las tablas utilizadas para conseguir este objetivo, que se muestran de forma gráfica en el modelo de la figura 6.1.

6.2.1. Usuarios, shares y permisos

En primer lugar, la base de datos debe permitir almacenar información sobre los usuarios, existentes así como los shares a los que estos pueden acceder. Para ello se utiliza una tabla de usuarios que contiene el nombre de login del usuario a demás de su identificador. La motivación de esta tabla (frente a usar directamente el nombre del usuario como clave) es utilizar pequeños identificadores de tamaño fijo como claves primarias, para minimizar el tamaño de los índices, así como facilitar el almacenamiento de otros datos del usuario, como puede ser el email de contacto. Además, se facilita el control de integridad, al no poder asociar shares ni permisos a usuarios inexistentes.

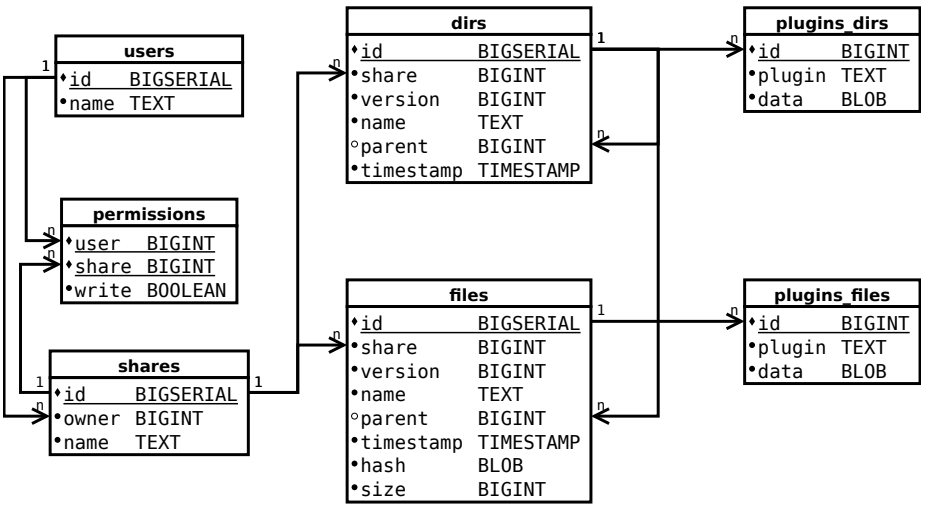


Figura 6.1: Base de datos del servidor

Los shares se almacenan en una tabla con otro identificador numérico único, conteniendo el nombre de la compartición y el identificador del usuario propietario. Existe una restricción de unicidad en el par de datos propietario-nombre, puesto que este es el identificador que utilizan los usuarios para añadir los shares, de manera que no tengan que memorizar su código numérico. La clave ajena utiliza actualización y borrado en cascada, de manera que al borrar un usuario se borran las comparticiones de las que es dueño.

Por último, la tabla `permissions` crea una relación N:M entre usuarios y comparticiones para permitir que usuarios accedan a shares de los que no son dueños. El atributo `write` se utiliza para diferenciar permisos de sólo-lectura frente a permisos totales. Las claves ajenas aseguran el borrado de los correspondientes permisos al borrar un usuario o una compartición.

6.2.2. Objetos: ficheros y directorios

La función principal de la base de datos es almacenar los metadatos de los objetos sincronizados, lo que se consigue con el par de tablas `files` y `dirs`. Esta división viene motivada por tener los ficheros un par de campos adicionales (el tamaño y el hash) y por el hecho de que el padre de un objeto siempre tiene que ser un directorio. La separación en dos tablas facilita mantener la segunda condición, puesto que la clave ajena de padre siempre se refiere a objetos de la tabla de directorios.

Además de los metadatos del fichero (nombre, fecha de última modificación y tamaño y hash en el caso de ficheros), se almacena en estas tablas un identificador único del objeto, la compartición a la que pertenece y la versión. Los identificadores son únicos a nivel global dentro de cada tabla, es decir, puede haber un directorio y un fichero con el mismo identificador, pero estos identificadores nunca se pueden repetir, ni siquiera en shares distintos.

El campo de la compartición evita el acceso a ficheros ajenos a las comparticiones a las que el usuario tiene acceso, y la versión se utiliza para diferenciar las

distintas revisiones a las que se somete un objeto. Se representa como un entero positivo, que comienza en 0 y se incrementa con cada cambio a los metadatos del objeto. Al cambiar los datos, los metadatos (el hash) también son modificados, así que cualquier tipo de cambio se refleja en este campo, incluyendo los cambios a los metadatos de plugins.

Estos metadatos se almacenan en una tabla adicional, que permite a los plugins almacenar información extra sobre los ficheros en un campo libre. Cada dato se identifica con el identificador del objeto y el nombre del plugin que lo genera. Para evitar redundancia, se omite el identificador de versión en estas tablas, puesto que en las tablas `dirs` y `files` habrá una única versión en un momento dado.

6.2.3. Datos históricos

Para el almacenamiento de datos históricos, se utilizan cuatro tablas adicionales, equivalentes a las de metadatos y plugins de ficheros y directorios. El uso de tablas separadas para los datos actuales y los históricos, permite almacenar todas las versiones de un dato y un gran rendimiento en las consultas a los datos actuales por mantenerse estos en una tabla separada y, por tanto, de menor tamaño.

Estas tablas, no reflejadas en el diagrama de la figura 6.1 son idénticas a las correspondientes con los datos actuales, a excepción de la clave primaria, que engloba tanto el identificador del objeto como su versión en el caso de las tablas históricas. Este cambio en la clave primaria implica añadir la versión a las tablas de plugins, de manera que puedan hacer referencia a la clave primaria de las tablas de metadatos.

El movimiento de datos de las tablas actualizadas a las históricas se puede realizar desde el programa, mediante sentencias SQL, o directamente en el motor de la base de datos utilizando disparadores. Las primera opción es compatible con cualquier base de datos, lo que la hace excelente como opción secundaria cuando los disparadores no están disponibles.

Para el prototipo, se utiliza PostgreSQL como sistema gestor de base de datos, ya que es probablemente el sistema gratuito y de código libre con más capacidades, entre las que se incluyen disparadores. Ya que se dispone de la capacidad, se utilizan disparadores para mover los datos entre las tablas, ya que al ejecutarse directamente en el motor de base de datos pueden beneficiarse de optimizaciones y garantías adicionales, además de evitar tener que gestionarlo manualmente desde el código. Para ello se utiliza el código (específico para PostgreSQL) mostrado en el listado de código 6.1 para actualizar los ficheros, siendo el código para los directorios prácticamente idéntico (sólo cambian los nombres de tablas).

6.2.4. Capa de acceso a datos

Para el acceso a la base de datos desde la aplicación se utilizan las clases `User` para los usuarios y permisos, `Share` para las comparticiones y `Dir` y `File` para los directorios y ficheros respectivamente, incluyendo la utilización de plugins. El programa servidor no tiene acceso directo a las tablas de datos históricos, puesto que los cliente no tienen un sistema para recuperar esta información.

```
CREATE OR REPLACE FUNCTION archive_file()
  RETURNS TRIGGER AS
  $BODY$
    BEGIN
      -- Aumentar número de versión en los nuevos datos
      NEW.version = OLD.version + 1;
      -- Archivar datos de plugins antiguos
      INSERT INTO files_plugins_old
        SELECT p.*, OLD.version FROM files_plugins AS p
          WHERE p.id = OLD.id;
      -- Archivar metadatos antiguos
      INSERT INTO files_old SELECT OLD.*;
      -- Actualizar la tabla de datos actuales
      RETURN NEW;
    END;
  $BODY$

CREATE TRIGGER archive_file
  BEFORE UPDATE
  ON files
  FOR EACH ROW
  EXECUTE PROCEDURE archive_file();
```

Listado de código 6.1: Trigger para auto-archivo de versiones antiguas de ficheros

Todas las clases utilizan unos métodos similares para acceder a la información: métodos `get*()` para acceder a un elemento concreto, `list*()` para acceder a un listado de elementos más amplio, `remove()` para borrar elementos y `save()` para guardar los cambios, tanto para inserciones como para modificaciones, lo cuál se detecta automáticamente.

6.3. Comparticiones: ServerConnectionShare

Esta clase gestiona los mensajes específicos de las comparticiones, modificando la base de datos y notificando a otros clientes según sea necesario. En todo caso, el primer mensaje procesado es el comando `InitShare`, que incorpora el árbol completo de ficheros en la compartición. Para ello se deserializa el árbol, y se compara con el que se encuentra en la base de datos. El primer paso es crear un árbol de nodos tipo `ShareTreeNode` o, si la compartición ya está abierta por otro cliente, acceder al ya existente.

A continuación, se utiliza el algoritmo mostrado en el listado de código 6.2 para detectar que objetos han sido creados, borrados, renombrados o modificados. Para ello se va recorriendo los datos serializados utilizando una pila para almacenar la ruta actual, y se van almacenando en tres listas los objetos borrados, modificados o creados.

Los ficheros renombrados se detectan al final del proceso, calculando la intersección entre las listas de borrados y creados buscando elementos coincidentes

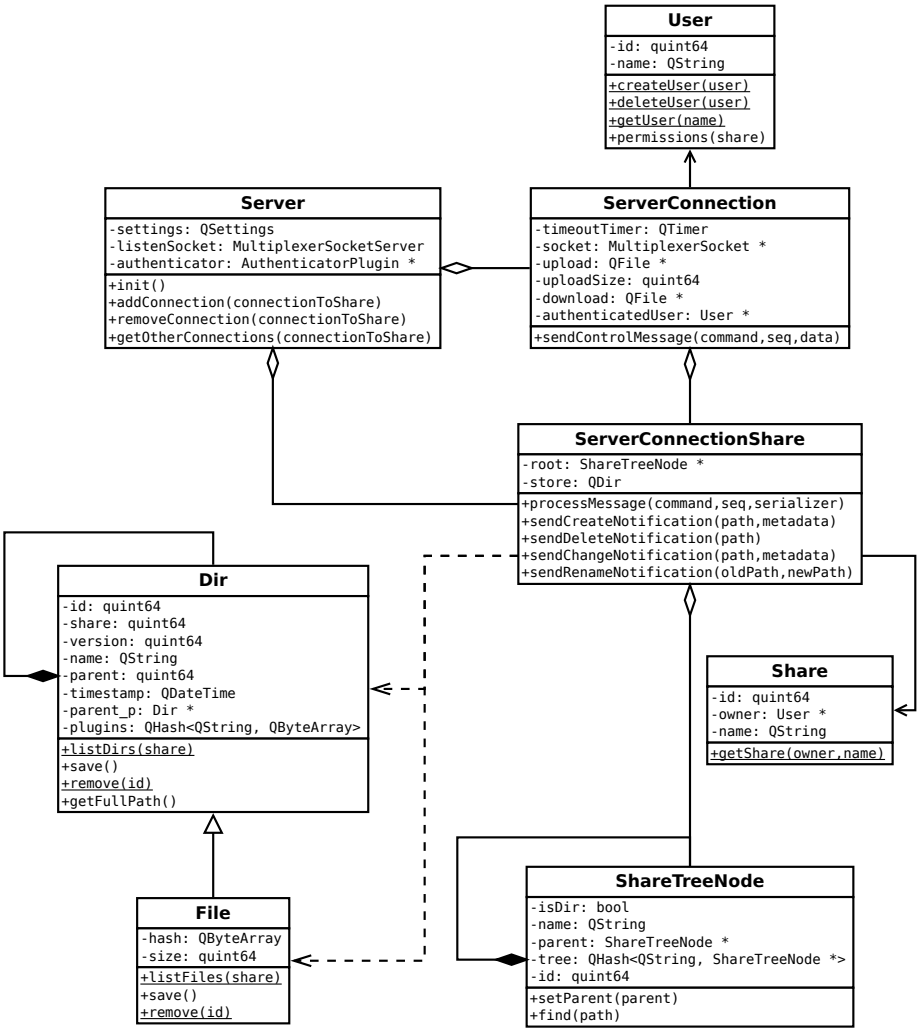


Figura 6.2: Diagrama de clases simplificado del servidor

en todo menos la ruta. Sólo se detectan ficheros renombrados, puesto que su detección es más sencilla y fiable (utilizando el hash) y son los que proporcionan la mayor ganancia (ahorro de transferencia de ficheros), mientras que los directorios pueden ser simplemente borrados y creados con otro nombre.

A continuación se envían las notificaciones. Para facilitar el trabajo al cliente, primero se mandan las creaciones, luego los cambios de nombre y modificaciones y finalmente los borrados, lo que permite que se creen los directorios destino para los ficheros renombrados y evita que estos ficheros sean borrados porque se borre su directorio contenedor.

Una vez superada la apertura del share, el comportamiento es muy sencillo. Simplemente se esperan órdenes del cliente y se actúa acordemente. Por ejemplo, si se solicita la creación de un objeto se comprobarán los permisos del usuario, la no existencia del objeto, etc. y se creará el objeto tanto en la base de datos como en el árbol de nodos `ShareTreeNode`, puesto que es precisamente esta estructura la que facilitará conexiones de otro clientes y validación de comandos (como la no existencia del objeto a crear antes mencionada) sin tener que acudir a la base de datos.

Finalmente, si el objeto ha sido creado, se enviará una notificación al resto de clientes conectados al share, utilizando el método `getOtherConnections()` de la clase `Server` (ver sección 6.5) para localizar las otras conexiones. En todo caso, se enviará un mensaje al cliente con el código de respuesta correspondiente.

6.4. Conexiones: ServerConnection

La gestión de las características propias de la conexión se realiza en esta clase. En el caso del servidor esto incluye el login y apertura de comparticiones, así como la transferencia de ficheros.

6.4.1. Autenticación y apertura de shares

La autenticación se delega al plugin de autenticación en uso, tal y como lo proporciona la clase del servidor. Simplemente se pasan las credenciales al plugin que verificará su validez, y se procederá a informar al cliente del resultado. En caso afirmativa a demás se almacena un objeto `User` de la capa de acceso a datos para conocer el identificador del usuario, necesario para posteriores operaciones.

La apertura de shares consiste en verificar que la compartición existe y el usuario tiene permisos para utilizarla a través del objeto `User` previamente almacenado y el objeto `Share` correspondiente a la compartición que se desea abrir. Si los permisos son correctos, se creará una instancia de la clase `ServerConnectionShare` que gestionará los sucesivos mensajes referidos a esta.

6.4.2. Transferencias y almacenamiento

Las subidas y descargas funcionan de forma simétrica a tal y como se describen en el cliente, en la sección 5.5.1. La principal diferencia se encuentra en los directorios de almacenamiento tanto de los ficheros finales como temporales, así como su nomenclatura, que utiliza los hashes de los datos como nombre de fichero.

```
while (ser->isOk()) { //Mas objetos serializados
  while (stack.size() > 0 && stack.top().first == 0) {
    //No quedan mas elementos en este nivel, los elementos
    //de la base de datos que no han sido procesados => nuevos
    [...]
  }

  //Leer siguiente objeto serializado
  [...]

  if (isDir) { //Objeto directorio
    [...]
    //Buscar elemento en el árbol de nodos del servidor
    while (filesIt.hasNext()) {
      [...]
      if (f->getName() == md.getName()) {
        //Mismo nombre, comparar versiones y fechas
        if (md.getVersion() == VERSION_INVALID) {
          if (md.getLastModified() != f->getTimestamp())
            changed << f;
        } else if (md.getVersion() != f->getVersion()) {
          changed << f;
        }
        [...]
        break;
      }
    }

    if (!found) {
      //No se encuentra en el árbol, borrado
      [...]
      deleted << dd;
    }

    if (found) {
      //Si el directorio existe, añadir a la pila
      stack.push([...]);
    } else if (childrenSize > 0) {
      //Si el directorio ha sido borrado y tiene hijos
      //deserializar e ignorar los hijos
      [...]
    }
  } else { //Objeto fichero
    [...]
    //Buscar elemento en el árbol de nodos del servidor
    while (filesIt.hasNext()) {
      [...]
      if (f->getName() == md.getName()) {
        //Mismo nombre, comparar versiones y fechas
        [...]
        if ((md.getVersion() == VERSION_INVALID &&
            md.getHash() != f->getHash()) ||
            (md.getVersion() != VERSION_INVALID &&
            md.getVersion() != f->getVersion())) {
          changed << f;
        }
        break;
      }
    }

    if (!found) {
      //No se encuentra en el árbol, borrado
      [...]
      deleted << dd;
    }
  }
}
```

Listado de código 6.2: Algoritmo para comparar árboles de directorio

En concreto, se almacenan todos los datos dentro de un directorio, con subdirectorios para cada compartición nombrados con su identificador numérico. Dentro de cada compartición se almacenan directamente los ficheros de datos nombrados con el hash de los datos que contienen además de un subdirectorio para ficheros temporales llamado `tmp`, utilizado para las subidas no completadas. Un pequeño ejemplo de esta estructura se muestra en el listado de código 6.3. La division en subdirectorios por cada compartición implementa el dominio de colisión por share, tal y como se discute en la subsección 9.1.2.

```
store
|— 10
|    |— 65178de662cd5a68daf679df0009695d6d682e01
|    \— tmp
\— 2
     |— 0592640e1ed0323e2b6ad2fbc8abf70bc7f1a8f5
     |— 3458426f0b402ee81e93037be412698278e23a1b
     |— a1e4aab80e037aa8dbf1ef6efa50292209916657
     \— tmp
          |— ddaf9fc9384c193134b40220628e235b8c8aa8f1
          \— fe4b2aa661547a7093460dd5687dbc6cb8d3ff9a
```

Listado de código 6.3: Ejemplo de árbol de directorios del servidor

Así, todas las subidas crearán inicialmente un fichero en el directorio temporal correspondiente al share en el que se almacenará finalmente el fichero. Si la subida falla o es abortada, el fichero será borrado, así como en cada arranque del servidor (para evitar la proliferación de ficheros temporales en caso de que el servidor sea cerrado en plena transferencia). En caso contrario, simplemente se moverá el fichero del directorio temporal al directorio principal de la compartición en cuestión.

6.4.3. Mantenimiento de la conexión

El mantenimiento de la conexión es muy sencillo desde el lado del servidor. Este simplemente tiene que responder a los comandos `Ping` con la respuesta `Ok` y la hora del sistema, puesto que será el cliente el encargado de calcular la diferencia entre las fechas según sea necesario.

Tal y como en el cliente, también se utiliza un temporizador para cerrar al conexión en caso de que no se reciban mensajes en un periodo de tiempo dado, actualmente de 30 segundos (el doble del intervalo entre pings del cliente). En ese caso, cerrará la conexión, y se cerrarán las comparticiones asociadas.

6.5. Funciones globales: Server

El último componente del servidor es la clase `Server`, ocupada de gestionar la configuración del servidor y los plugins, además de facilitar la interacción entre los distintos clientes utilizando la misma compartición.

Esto último se consigue almacenando en un mapa cuya clave es el identificador de cada share, una listado de conexiones abiertas a esa compartición, de

forma que cuando una de estas procesa un cambio, puede consultar a la clase `Server` para obtener el listado del resto de conexiones a las que debe enviar la notificación.

Para al configuración y la carga de plugins se utilizan los mismos sistemas descritos en el cliente, en las subsecciones 5.6.1 y 5.6.2. Sin embargo, los datos almacenados en la configuración y los plugins cargados son distintos.

Los parámetros configurables en el servidor son:

- La conexión a la base de datos (clave `db`) incluyendo el driver a utilizar, y todos los parámetros necesarios para la conexión: servidor, puerto, usuario, contraseña y base de datos.
- El directorio de almacenamiento de ficheros: `files/store`.
- El socket en el que escuchar la conexión: `socket/host` y `socket/port` que toman por defecto los valores 0.0.0.0 (indicando todas las interfaces) y 48135.
- Los plugins a utilizar bajo la clave `plugins` y subclave equivalente al tipo de plugin. Además de la subclave `dir` para indicar la ruta del directorio de plugins.

Los únicos valores obligatorios y que no toman valor por defecto son los dos primeros: base de datos y directorio de almacenamiento, para lo que se dispone de una pequeña interfaz por consola para configurarlos, pasando la opción `--configure` al ejecutable del servidor. También se proporcionan opciones para ver y cambiar opciones individuales, documentadas en el propio ejecutables cuando recibe parámetros inválidos o la opción `--help`.

Entre los plugins, el único obligatorio es el de autenticación, que se selecciona de la configuración o, si hay uno sólo disponible, de forma automática. Este tipo de plugins proporciona una forma de integrar el sistema con cualquier sistema de autenticación preexistente, mediante la implementación de un método que recibe las credenciales del usuario y devuelve si la autenticación es correcta. Los detalles se encuentran en el capítulo correspondiente a los plugins, en la sección 7.3.

Capítulo 7

Plugins

En este capítulo se detalla el diseño del sistema de plugins y de cada uno de ellos. En primer lugar se describen las características fundamentales de todos los plugins, así como el sistema utilizado para cargar estos plugins en el sistema.

A continuación se describen los plugins para los que se ha desarrollado una implementación, incluyendo tanto la interfaz del plugin como el diseño de la implementación en sí. Esto incluye los dos plugins obligatorios: monitoreo de ficheros y autenticación, además de una interfaz gráfica para el cliente de escritorio.

Finalmente se tratan el resto de interfaces de plugins, aquellas que pueden utilizarse pero para las que no se ha creado ninguna implementación, quedando esto totalmente a cargo de los usuarios del sistema o como trabajos futuros.

7.1. Características comunes

Todos los plugins del sistema, deben tener la posibilidad de almacenar valores de configuración, puesto que muchos de ellos lo utilizarán para memorizar las preferencias del usuario. Para facilitar esta labor sin que el plugin deba implementar su propio sistema, se utiliza el ya conocido sistema de Qt, tal y como se describe en la subsección 5.6.1. Sin embargo, este sistema permite acceso total a los ficheros de configuración, con lo que el plugin podría modificar accidentalmente datos ajenos.

Para evitar este problema sin recurrir a la utilización de distintos ficheros de configuración por plugin, se crean unas funcionalidades de acceso a la configuración comunes a todos los plugins, residentes en la clase `BasePlugin`, cuya interfaz se muestra en el listado de código 7.1.

Todos los plugins heredan de esta clase y por tanto disponen de la funcionalidad que esta provee. Dado que se desea facilitar el acceso a la configuración, se mantiene una referencia al objeto `QSettings` de la aplicación además de un prefijo, consistente en el nombre del plugin, precedido de la cadena fija `plugincfg/`. Además provee métodos equivalentes a los de la clase `QSettings` para almacenar y recuperar datos de la configuración, a los que automáticamente añade el prefijo para acceder sólo a los datos propios del plugin.

Las clases que implementan los plugins no heredan de forma directa de `BasePlugin` si no que heredan de la interfaz que implementan que a su vez

```
class BasePlugin {
public:
        void init(QSettings *set, QString prefix);
        QVariant sValue(QString key, QVariant
            defaultValue = QVariant());
        bool sContains(QString key);
        void sSetValue(QString key, QVariant value);

private:
        QString prefix;
        QSettings *settings;
};
```

Listado de código 7.1: Clase base para todos los plugins

hereda de `BasePlugin` tal y como se muestra en la figura 7.1. Todo plugin también debe heredar directa o indirectamente de la clase `QObject` incluida en Qt, siendo esto un requisito para la utilización del sistema de plugins de dicha librería. La clase `QObject` añade metainformación sobre la clase, lo que permite identificar el plugin y la interfaz que utiliza, así como su versión.

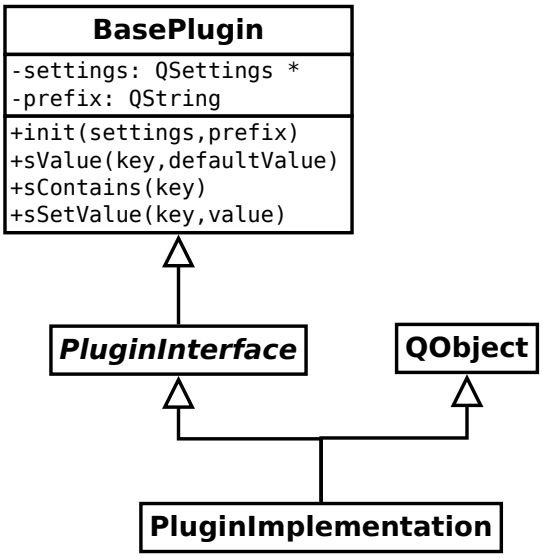


Figura 7.1: Diagrama de herencia de plugins

Para cargar plugins se utiliza la clase `QPluginLoader` que se ocupa de, dada la ruta de un fichero, cargarlo en memoria como librería dinámica, identificar las interfaces que implementa (mediante los metadatos de `QObject`) y devolver una instancia. El programa comprobará las instancias cargadas y el fichero de configuración para elegir que plugin cargar y, a continuación, llamará a la función `init()` del plugin para darle acceso a la configuración y almacenará las instancia en las estructuras de datos necesarias.

Ya que los plugins se cargan como librerías dinámicas, se pueden realizar llamadas entre el programa principal y el plugin simplemente como llamadas a métodos, con lo que el programa principal podrá notificar al plugin de eventos llamando a los métodos correspondientes o emitiendo una señal que se conectará al slot del plugin, puesto que un slot no es más que un método con metadatos añadidos. De igual forma, el plugin puede acceder a los métodos del programa principal siempre que se le pase una referencia a las instancias de las clases que debe manipular.

Al ser un plugin una librería, es decir, un conjunto de código objeto, no hay ningún problema en que el mismo plugin implemente varias interfaces en varias clases, y se comuniquen entre sí. Esto permite la composición de comportamientos más complejos en los plugins mediante la utilización de múltiples funcionalidades expuestas a través de las interfaces de los plugins.

7.2. Monitoreo del sistema de ficheros

Los plugins para el monitoreo del sistema de ficheros deben permitir añadir y borrar directorios a monitorizar, según se añadan o quiten puntos de montaje en el cliente. Mientras un directorio esté monitorizado, todo cambio en el directorio o cualquiera de sus descendientes (el monitoreo debe ser recursivo) debe ser detectado e informarse al cliente de este cambio.

La creación de este tipo de plugins está motivada por el requisito del sistema de ser multiplataforma y la ausencia de un sistema unificado de monitoreo del sistema de ficheros a excepción del ineficiente re-escaneo continuo de los puntos de montaje. A pesar de esto, la primera opción fue utilizar un sistema de monitoreo incluido de serie en Qt, implementado en la clase `QFileSystemWatcher` [24] que utiliza las APIs específicas de cada sistema operativo.

Sin embargo, la utilización de este sistema fue imposible debido a graves deficiencias para el uso deseado. En concreto, Qt sólo notifica de cambios en directorios y ficheros de forma independiente, de manera que es necesario monitorizar cada objeto por separado. En las pruebas, el consumo de recursos que esto supone excedía las capacidades de los sistemas operativos, que sencillamente denegaban la operación.

El resto de opciones multiplataforma para el monitoreo de ficheros estaban integradas en librerías multipropósito, que aumentaban mucho el tamaño de los ficheros a distribuir con el sistema, además de los tiempos de compilación. Debido a la ausencia de soluciones, se decidió implementar directamente una solución específica para el prototipo, que además puede beneficiarse de optimizaciones específicas para el uso deseado. La necesidad de extraer esta funcionalidad como plugin, surge evita añadir código específico para cada plataforma al núcleo del programa, manteniéndose neutral.

La implementación de este plugin es crítica, puesto que es necesario para el funcionamiento del cliente, siendo necesarias implementaciones distintas según el sistema operativo utilizado por las diferencias entre sus APIs. El prototipo incluye plugins funcionales para Windows y Linux, y código incompleto para Mac OS X, que no ha podido ser compilado ni probado por no disponer del hardware necesario.

7.2.1. Interfaz

En la interfaz mostrada en el listado de código 7.2, se dispone de:

- Un destructor virtual para permitir que el plugin cierre sus recursos al descargarse.
- Los métodos `addWatch()` y `removeWatch()`, que reciben la ruta al directorio que debe empezar o dejar de monitorizarse. Afectan a la ruta pasada como parámetros y todos sus descendientes, directos e indirectos.
- Un comentario indicando las señales que deben generar estos plugins. Las rutas de las señales deben ser absolutas.

Como se puede observar, el método elegido para enviar las notificaciones son las señales de Qt, teniendo esto el problema de que no pueden aparecer en la interfaz, puesto que no esta no hereda de `QObject`, siendo imposible hacer que la interfaz herede por el funcionamiento del sistema de plugins de Qt. Sin embargo, aunque estas comprobaciones o pueden hacerse en tiempo de compilación como los métodos virtuales puro normales, pueden realizarse y se realizan en tiempo de carga del plugin, puesto que Qt puede detectar la no existencia de estas señales al llamar al conectar las señales a sus slots correspondientes, en cuyo caso el plugin no se cargará correctamente.

Otra posibilidad que tiene la ventaja de poder verificarse durante la compilación sería utilizar una clase auxiliar que contenga los métodos correspondientes a las señales, y de la cuál se creara una instancia en el núcleo del cliente pasando una referencia al plugin para que llamara a dichos métodos. Sin embargo, esto implica la creación de una clase adicional que formaría parte de la interfaz del plugin, además de no aprovechar las capacidades para mensajería multi-hilo de las señales de Qt.

La elección final viene dada por el buen funcionamiento a través de hilos de ejecución de las señales de Qt, ya que tras analizar la documentación de las soluciones para el monitoreo de ficheros en los distintos sistemas operativos, todas ellas necesitan o se benefician de la ejecución en un hilo para recibir los eventos del sistema operativo.

7.2.2. Plugin para Linux: inotify

En Linux, se utiliza *inotify* (que reemplaza a *dnotify*) para realizar el monitoreo. Esta interfaz está disponible desde la versión 2.6.12 del kernel (Agosto de 2005) hasta la actualidad, por lo que no parece necesario implementar también un plugin para *dnotify*, cuya única utilidad sería soportar equipos con Linux muy antiguos, que además probablemente no soportaran otras librerías usadas en el proyecto.

La documentación de *inotify* puede encontrarse en el fichero `inotify.txt` [25] suministrado con el kernel y en las páginas *man* correspondientes a cada función utilizada, sirviendo `man inotify` como punto de entrada. A continuación se comentan las características principales.

Introducción a inotify

Como la mayoría de APIs del mundo Unix, *inotify* utiliza objetos que se comportan como ficheros para proporcionar su funcionalidad. La inicialización

```
// All paths (methods and signals) are absolute
class FSWatcherPlugin : public BasePlugin {
public:
        virtual ~FSWatcherPlugin() {}

        virtual bool addWatch(QString path) = 0;
        virtual void removeWatch(QString path) = 0;

/* Use the following signals for notifications
signals:
        void created(QString path);
        void changed(QString path);
        void deleted(QString path);
        void renamed(QString from, QString to);
        void error();
*/
};

Q_DECLARE_INTERFACE(FSWatcherPlugin, "bsync.client.
    fswatcher/1.0");
```

Listado de código 7.2: Interfaz para los plugins de monitoreo del sistema de ficheros

del sistema con `inotify_init()` crea un descriptor de ficheros que se puede utilizar con la funciones `inotify_add_watch()` y `inotify_rm_watch()` para iniciar o detener el monitoreo sobre un directorio dado.

Una vez iniciado el monitoreo, los eventos se reciben a través del descriptor devuelto por la llamada a `inotify_init()`, utilizando la misma función que con cualquier otro fichero, es decir, `read()`. Los datos leídos corresponden con una estructura de C, que contiene el directorio sobre el que se ha detectado el cambio (identificado por un entero devuelto por `inotify_add_watch()`), una máscara de bits indicando los eventos detectados y el nombre del fichero o directorio afectado. Además, se incluye un identificador opaco o cookie utilizado para relacionar eventos entre sí, especialmente cambios de nombre que se dividen en un evento con el nombre antiguo y otro con el nuevo nombre.

El monitoreo solamente afecta a un directorio y a sus hijos directos, de manera que si se quiere monitorizar un árbol, es necesario llamar a la función `inotify_add_watch()` una vez por cada directorio en el árbol, lo que puede exceder los recursos asociados al sistema de *inotify*, normalmente de 150.000 directorios para todo el sistema.

Funcionamiento del plugin: InotifyFSWatcher

El plugin se implementa en la clase `InotifyFSWatcher`, que se ejecuta en un hilo de ejecución propio para evitar que las espera de eventos en al función `read()` bloquee el resto del sistema.

Dado que la interfaz del plugin requiere que se monitoreen subárboles com-

pletos en vez de un simple directorio, se utiliza la técnica mencionada en la subsección anterior, en la que se llama a `inotify_add_watch()` sobre cada directorio del subárbol, almacenando los identificadores numéricos de los directorios en un mapa que los asocia a su ruta completa, para poder reconstruir la ruta completa de las notificaciones recibidas.

Sólo se atiende a un pequeño conjunto de los eventos proporcionados por *inotify*: borrados, creaciones, modificaciones de datos y metadatos, cierre de ficheros en modo de escritura y cambios de nombre, además de las notificaciones de errores en el monitoreo.

Las creaciones y borrados se notifican inmediatamente, al igual que las modificaciones de los metadatos. Sin embargo, los cambios en los datos se notifican sólo en dos situaciones: al cerrar el fichero (indicando que no se van a hacer más cambios momentáneamente) o tras cierto tiempo (3 segundos) puesto que algunos programas modifican los ficheros sin cerrarlos al acabar la modificación. Para minimizar el número de temporizadores usados, se utiliza un único temporizador para todos los objetos monitorizados, en vez de uno por cada fichero modificado, lo que provoca que el tiempo de espera tras la modificación pueda ser inferior a los tres segundos.

Para los cambios de nombre es necesaria una lógica un poco más complicada, puesto que esto también incluye movimientos hacia dentro o fuera del punto de montaje monitorizado y además se notifica como dos eventos consecutivos. Si reciben dos eventos `IN_MOVED_FROM` y `IN_MOVED_TO` de forma consecutiva y con la misma cookie, se interpreta como un cambio de nombre. Sin embargo, la recepción de `IN_MOVED_TO` sin el correspondiente `IN_MOVED_FROM` corresponde a una creación y el caso contrario a un borrado.

7.2.3. Plugin para Windows: ReadDirectoryChangesW

Windows utiliza la misma interfaz para el monitoreo de sistemas de ficheros desde Windows 2000 hasta la actualidad, la función `ReadDirectoryChangesW()`, documentada en MSDN en [26].

Introducción a ReadDirectoryChangesW

El acceso a todo el subsistema de monitoreo de directorios se realiza a través de una única función: `ReadDirectoryChangesW()`. La primera llamada a esta inicia el sistema y los buffers necesarios, mientras que sucesivas llamadas con el mismo directorio devuelven los eventos detectados en el sistema de ficheros.

Toda la información se devuelve en un único buffer pasado por parámetros a la función, en el que se incluye el puntero al siguiente evento (o longitud de este evento), el tipo de cambio ocurrido y la ruta al objeto afectado relativa al directorio monitorizado.

Existe también un parámetro utilizado para indicar si se desea monitorizar el directorio y todos sus descendientes de forma recursiva o no, en cuyo caso sólo se notifica de cambios en los hijos directos. También se incluyen parámetros para permitir el monitoreo asíncrono, en cuyo caso la función devolverá automáticamente y se informará del cambio cuando ocurra a través de uno de otros tres sistemas.

Funcionamiento del plugin: Win32FSWatcher

A pesar de que el funcionamiento del API de Windows parece sencillo al contenerse en una única función, en la práctica fue el más complicado de implementar de los tres plugins, debido a la necesidad de utilizar el método asíncrono de recepción de notificaciones para poder recibir eventos de distintos directorios. La implementación final está basada en las implementaciones de plugins similares para las librerías de Qt, wxWidgets y JNotify [27].

Todo el código se ejecuta en un hilo separado del programa principal, para evitar que interfieran entre sí, principalmente debido a que el plugin requiere latencias pequeñas para funcionar correctamente, puesto que las colas de eventos de Windows son relativamente pequeñas y en caso de desbordarse se pierden todos los eventos acumulados en estas. Por ello, se recogen los eventos en cuanto se recibe la notificación asíncrona de que se ha producido, copiándolo a una cola de eventos local en el plugin.

Siempre que no se están recibiendo eventos del sistema, se comprueba la cola local y se procesan los eventos presentes en ella, notificando al programa principal de forma directa, al existir una correspondencia directa entre los eventos de Windows y las señales del plugin. De hecho, la única modificación al evento enviado por el sistema operativo es añadir la ruta del directorio monitorizado, puesto que Windows trabaja con rutas relativas y el plugin debe enviar rutas absolutas.

7.2.4. Plugin para Mac OS X: FSEvents

Mac OS X dispone de dos sistemas para la notificación de eventos del sistema de ficheros *FSEvents* y las colas del kernel o *kqueue*s. Mientras que el primero monitorea subárboles de directorios, informando sólo de la existencia de cambios (pero no de sus detalles), el segundo monitorea un único fichero notificando de cualquier evento relacionado con él.

Obviamente la situación ideal sería un sistema que monitoree subárboles enteros, notificando de todos los cambios, pero entre las dos alternativas disponibles, la más adecuada es *FSEvents* por permitir monitorizar gran cantidad de objetos con un único descriptor, ya que los descriptores de *kqueue* son limitados y probablemente inferiores a los necesarios para los propósitos de este sistema, además de consumir más recursos.

FSEvents está disponible en todas las versiones de Mac OS X desde la 10.5, es decir, desde Octubre de 2007, por lo que debería poder utilizarse en cualquier ordenador de Apple relativamente moderno. El API se encuentra bien documentado en la página para desarrolladores de Apple [28].

Introducción a FSEvents

El API de *FSEvents* se inicializa llamando a `FSEventStreamCreate()` por cada subárbol de directorios a monitorizar. Esta función recibe como parámetro un puntero a una función que se ejecutará cada vez que se reciba algún evento en ese directorio. A continuación se llama a un par de funciones más para iniciar el bucle de eventos y solicitar el comienzo de envío de eventos.

Cada evento sólo contiene la ruta donde se ha producido el evento, que puede ser el directorio monitorizado o alguno de sus descendientes. Eventos cercanos

en el tiempo se agregan en uno sólo, referido al primer directorio padre que tengan en común los eventos, de manera que a menudo se recibirán eventos para el directorio superior, a pesar de que los cambios se han efectuado sobre objetos a mayor profundidad en el árbol del sistema de ficheros.

Funcionamiento del plugin: FSEventsFSWatcher

La idea básica en este plugin es re-escanear los directorios afectados cada vez que se produce un evento, para lo cuál es necesario mantener en memoria la información básica de todos los objetos monitorizados, lo que incrementa en gran medida el consumo de RAM de este plugin respecto al resto.

Cada vez que se completa un escaneo, se utiliza un algoritmo similar al de inicialización de shares del servidor, descrito en la sección 6.3. La principal diferencia es la capacidad de utilizar el número de inodo como identificador único de los objetos, lo que permite detectar cambios de nombre de manera sencilla.

7.3. Autenticación

Separar la autenticación en un plugin permite una fácil integración con sistemas de autenticación preexistentes, de manera que no sea necesario mantener credenciales distintas para distintos servicios. Dada la gran variedad de los sistemas de autenticación, la creación de un plugin es la mejor opción, puesto que permite la utilización de un backend arbitrario.

La autenticación no es la única tarea de estos plugins, que también deben permitir todas las tareas relacionadas con la gestión de credenciales: crear y borrar usuarios y cambiar contraseñas. Dependiendo del backend utilizado, esto puede no ser posible, por ejemplo en sistemas empresariales en los que el cambio de contraseñas para todos los servicios deba realizarse desde una interfaz determinada. Por lo tanto, los plugins deberán soportar backends que soporten o no estas operaciones de modificación.

Puesto que este plugin es imprescindible para el funcionamiento del servidor, se incluye una implementación que almacena las contraseñas en la misma base de datos que se utiliza para almacenar los metadatos de objetos sincronizados, utilizando el módulo crypto de PostgreSQL para su almacenamiento seguro.

7.3.1. Interfaz

La interfaz final para los plugins de autenticación se muestra en el listado de código 7.3. El método principal es `authenticate()` que toma un usuario y contraseña y devuelve si estos son correctos.

También se dispone de métodos para crear, borrar y modificar credenciales, cuya implementación es opcional, siempre que `isEditable()` devuelva `false`. En este caso, al crear un usuario en el sistema, se verificará su existencia utilizando el método de autenticación, de forma que sólo se permitan crear usuarios preexistentes en el backend de almacenamiento de credenciales. Esto permite la creación automática de cuentas para el sistema, siempre que ya existieran en el backend.

```
class AuthenticatorPlugin : public BasePlugin {
public:
        virtual bool authenticate(QString user, QString
            password) = 0;

        virtual bool createPassword(QString user, QString
            password) = 0;
        virtual bool changePassword(QString user, QString
            password) = 0;
        virtual bool deletePassword(QString user) = 0;

        virtual bool isEditable() = 0;
};

Q_DECLARE_INTERFACE(AuthenticatorPlugin, "bsync.server.
    authenticator/1.0");
```

Listado de código 7.3: Interfaz para los plugins de autenticación

En caso de implementar estas operaciones, `isEditable()` devolverá `true`, y en caso de creación de cuenta se creará la credencial asociada utilizando `createPassword()`.

7.3.2. Implementación: PgCryptoAuthenticator

Para el prototipo se ha implementado un plugin que almacena las contraseñas en la base de datos del servidor, haciendo uso de una tabla creada a tal efecto, de nombre `pgcryptoauthenticator_users`. Esta tabla se compone de dos campos: el identificador o nombre de usuario, y un campo para almacenar la contraseña con las medidas de seguridad adecuadas, esto es, hasheadas y utilizando un *salt*, como se discuten a fondo en la subsección 9.2.2.

El nombre de estos campos es por defecto `id` para el usuario y `hash` para la contraseña, aunque junto con el nombre de la tabla puede configurarse mediante el uso del fichero de configuración. Esto hace que el plugin sea mucho más reutilizable, pudiendo utilizarse con cualquier esquema de base de datos, incluso algunos preexistentes, siempre que se utilice un esquema basado en la función `crypt()` de Unix para el almacenamiento de la contraseña. Dado que `crypt()` soporta múltiples algoritmos, se permite configurar el algoritmo utilizado para la creación de cuentas nuevas, pero la autenticación puede realizarse con éxito utilizando cualquiera de ellos.

El plugin soporta todos los métodos de la interfaz, incluyendo la creación, borrado y modificación de credenciales, devolviendo por tanto `true` a través del método `isEditable()`.

7.4. Interfaz de usuario

Para facilitar el uso del cliente a usuarios menos técnicos, se diseña un plugin que permita la creación de interfaces de usuario para comprobar y manipular

el estado del sistema. Puesto que cualquier tarea debería poder ser controlada desde esta interfaz, el plugin tendrá acceso a toda la información del cliente.

Este acceso total puede causar problemas de funcionamiento del sistema, ya que un error en el plugin podría modificar una parte crítica del estado del programa ocasionando su mal funcionamiento. Para evitar esto en la medida de lo posible, mientras que se mantiene acceso a todo el sistema se ha diseñado una interfaz que permite acceder a la mayor parte de la información del cliente de manera segura.

Además, el núcleo del cliente no tiene dependencias con librerías gráficas, puesto que toda su comunicación con el usuario se realiza a través de ficheros de configuración y salida de texto. Puesto que esto es una característica deseable, se debe evitar que se carguen las librerías gráficas en caso de no necesitarse. Conseguir esto es más complicado de lo que parece y se discute en su propia sección.

Para facilitar la introducción del sistema, se incluye un pequeño plugin de ejemplo. Este sólo muestra el estado global del sistema (parado, sincronizando o error) y permite configurar las conexiones a los servidores o mostrar los mensajes generados por la aplicación.

7.4.1. Carga dinámica de librerías gráficas

Para minimizar el número de dependencias en sistemas sin interfaz gráfica, como pueden ser los servidores Unix, se desea evitar la cargar de las librerías gráficas (el módulo QtGui) en caso de que no se encuentra activado ningún plugin que las necesite.

Para conseguir esto no debe haber ninguna referencia a clases del módulo gráfico de Qt en el cliente, lo cuál no parece en principio un problema puesto que estas clases sólo deberían utilizarse en los plugins gráficos. Sin embargo, Qt requiere que toda aplicación gráfica instancie un objeto de la clase `QApplication` y toda aplicación de consola una de la clase `QCoreApplication`, no facilitando ningún sistema por el que se pueda cambiar el tipo de aplicación.

Dado que esta clase es un requisito para el uso de muchas clases de Qt, incluyendo la lectura de los parámetros de la línea de comandos y de la configuración, debe crearse una instancia de uno de los dos tipos de aplicación antes de cargar el plugin gráfico, puesto que es necesario acceder a la configuración para decidir el plugin gráfico a cargar.

Para evitar añadir parámetros en tiempo de compilación y no perder la capacidad de seleccionar el plugin gráfico a través de la configuración, se decide utilizar un pequeño plugin que se intentará cargar al arranque de la aplicación. En caso de que se cargue con éxito, se llamará al método `createApplication()` del plugin que devolverá un objeto de tipo `QApplication` y en caso contrario el programa principal creará un objeto `QCoreApplication` y omitirá el paso de carga del plugin gráfico.

Dado que `QApplication` es una subclase de `QCoreApplication`, el núcleo del cliente podrá manejar adecuadamente el objeto creado, independientemente de que se haya creado internamente o por el plugin, permitiendo un arranque correcto del programa sin dependencias hacia la librería gráfica.

```
class GuiPlugin : public BasePlugin {
public:
        virtual void initGui(QAbstractItemModel *model,
            QSettings *settings) = 0;
};

class DirectGuiPlugin : public BasePlugin {
public:
        virtual void initGui(Client *client) = 0;
};

Q_DECLARE_INTERFACE(GuiPlugin, "bsync.client.gui/1.0");
Q_DECLARE_INTERFACE(DirectGuiPlugin, "bsync.client.
    directgui/1.0");
```

Listado de código 7.4: Interfaz para los plugins de autenticación

7.4.2. Interfaz

Para evitar que el plugin tenga acceso total a las clases del cliente, toda la comunicación se lleva a cabo a través de un modelo de datos que representa todo el conocimiento de la aplicación. De esta forma se utiliza el patrón de diseño modelo-vista, en el que el núcleo del programa expone un modelo de datos que se muestra en el plugin de interfaz gráfica, que actúa como vista.

Para ello se utilizan las clases destinadas a implementar este patrón proporcionadas por Qt, siendo el modelo una subclase de `QAbstractItemModel`. Esta clase abstracta de Qt simplifica la creación de modelos de datos, siendo sólo necesario implementar cuatro funciones de acceso a datos para conseguir un modelo de sólo-lectura, y un par de ellas más para permitir la escritura, implementado Qt el resto de funciones para navegar el modelo.

Cada elemento de un modelo de datos de Qt se identifica por el padre y un par de coordenadas (x e y). Esto permite la implementación de listas (utilizando sólo la x), tablas (x e y) y árboles (las tres) de manera sencilla. El modelo de datos del cliente sigue una estructura en forma de árbol, siendo sus niveles los siguientes:

1. **Estado global del cliente:** Estadísticas globales y plugins cargados.
2. **Conexiones:** Colas de transferencia de ficheros, estado de red, etc.
3. **Comparticiones:** Estado de los objetos monitorizados, tareas pendientes, etc.
4. **Directorios y ficheros:** Metadatos, estado de sincronización y transferencia, etc.

Cabe destacar que el estado global del cliente contiene únicamente algunas estadísticas resumen como el número de conexiones activas o la cantidad de ficheros pendientes de cambios, pero no incluye la configuración. Para facilitar el acceso y modificación de esta, en vez de incluirse en el modelo de datos, se proporciona un puntero a la clase `QSettings` para acceder de forma directa.

Dado que todas las operaciones se realizan sobre el modelo de datos, la interfaz del plugin es muy sencilla, tal y como se muestra en el listado de código 7.4. También se incluye una interfaz alternativa que proporciona acceso directo a las clases del cliente en caso de que el modelo de datos no incluya la información necesaria, pero se recomienda utilizar la primera siempre que sea posible.

7.4.3. Implementación: MiniGui

Puesto que la interfaz de usuario es sólo un objetivo secundario del proyecto, con el propósito de facilitar la puesta en marcha del sistema a usuarios no técnicos, la implementación desarrollada sólo cubre las necesidades mínimas: permitir la configuración e informar del estado.

El elemento principal de la interfaz es el icono de bsync, situado en la bandeja del sistema y con el aspecto mostrado en la figura 7.2. Este icono es el punto de acceso al resto de la interfaz, lo que se consigue con el menú que se muestra al hacer click derecho sobre este. Además, muestra de forma rápida el estado de la aplicación mediante la utilización de tres colores: verde, amarillo y rojo indicando reposo, sincronización en progreso y error respectivamente.

(a) Reposo (b) Sincronizando (c) Error

Figura 7.2: Iconos de estado de la interfaz gráfica

En caso de producirse un error en el sistema, se puede comprobar el error producido consultando los mensajes del sistema seleccionando la opción del registro de mensajes del menú contextual. Esto muestra un diálogo como el de la figura 7.3, con un listado de mensajes generados por el sistema en tres niveles de severidad: *debug* o informativos, *warning* o advertencia y *critical* o críticos. Existe un cuarto nivel, *fatal*, pero los mensajes de este tipo ocasionan el cierre inmediato de la aplicación, por tratarse de errores irrecuperables.

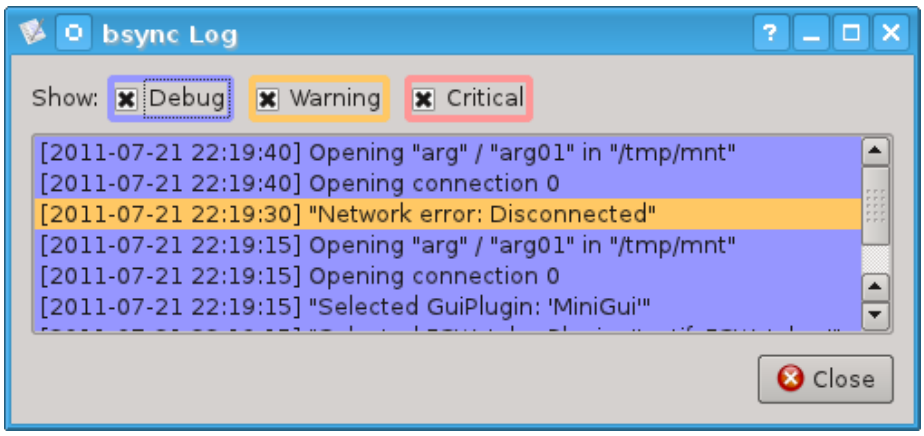


Figura 7.3: Listado de mensajes en la interfaz gráfica

Gracias a la utilización del sistema de mensajes de Qt, es trivial interceptar todos los mensajes generados por la aplicación llamando a la función `qInstallMsgHandler()`, que recibe como parámetro un puntero a una función que se ocupará de procesar los mensajes, en este caso, añadiéndolos al listado de la interfaz.

La lista permite el filtrado de mensajes según su nivel de severidad, que se muestra en la lista con un código de colores: azul para información, naranja para advertencias y rojo para errores. La configuración del filtrado se almacena automáticamente utilizando los métodos de `BasePlugin`, de manera que el usuario no tiene que cambiar los filtros cada vez que consulta el registro de mensajes.

El resto de la configuración se gestiona directamente utilizando la instancia de `QSettings` del cliente, a la que se puede acceder a través del puntero recibido en la inicialización. La interfaz gráfica desarrollada sólo permite configurar las conexiones a distintos servidores y shares que se utilizarán, utilizando la ventana mostrada en la figura 7.4.

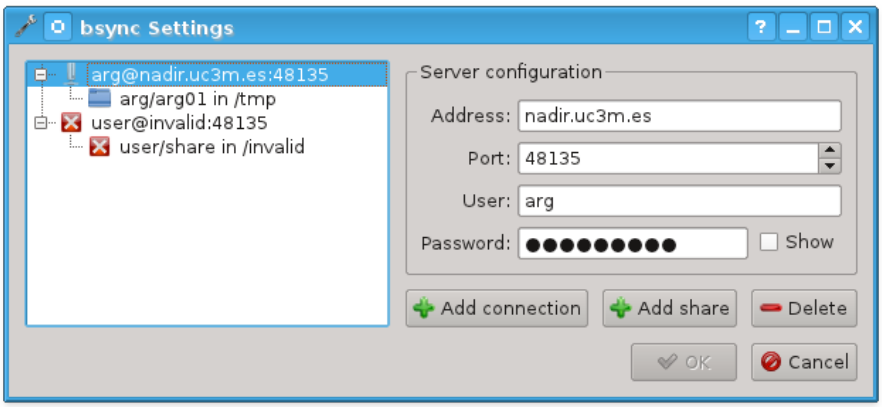


Figura 7.4: Diálogo de configuración de la interfaz gráfica

Todas las conexiones configuradas se muestran en la parte izquierda de la pantalla en forma de un árbol de dos niveles: conexiones y comparticiones. Al seleccionar un elemento, la parte derecha cambia para permitir configurar los campos apropiados para las conexiones (dirección y puerto del servidor, usuario y contraseña) y las comparticiones (propietario del share, nombre y punto de montaje).

Cualquier error se muestra cambiando el icono del elemento en la lista por una cruz roja y desactivando el botón de guardar cambios, para evitar el almacenamiento de configuraciones erróneas. Sólo se comprueba que los parámetros cumplen los requisitos mínimos, no comprobándose la conexión al servidor o las credenciales.

Para facilitar la internacionalización del producto, todas las cadenas de texto se han marcado de forma que sean reconocidas por el sistema de traducción de Qt, lo que permitirá traducir los textos de la interfaz (y de los mensajes generados por el programa) a cualquier idioma utilizando la herramienta Qt Linguist, que genera ficheros de lenguaje que reemplazan las cadenas de texto de todo el programa.

7.5. Otros plugins

En esta sección se describen el resto de interfaces de plugins soportadas por el sistema, pero para las cuáles no se ha creado ninguna implementacion de ejemplo. Muchos de estas interfaces son muy sencillas, y están diseñadas de manera que un plugin pueda utilizar varias de ellas para implementar la funcionalidad deseada.

7.5.1. Modificación de metadatos

Este plugin permite añadir metadatos específicos del plugin a los objetos sincronizados. Puede utilizarse en solitario, por ejemplo para incorporar etiquetas o de forma conjunto con otros plugins que necesiten almacenamiento persistente de información correspondiente a los objetos.

La interfaz se muestra en el listado de código 7.5, y se compone de un simple método que tiene acceso a todos los metadatos del objeto a sincronizar, y puede añadir información arbitraria de tipo `QVariant` que será enviada al servidor y por tanto a otros clientes.

```
class MetadataPlugin : public BasePlugin {
public:
        virtual QVariant getMetadata(CommonMetadata *md)
            = 0;
};

Q_DECLARE_INTERFACE(MetadataPlugin, "bsync.client.
    metadata/1.0");
```

Listado de código 7.5: Interfaz para el plugin de modificación de metadatos

7.5.2. Modificación de datos

La interfaz mostrada en el listado de código 7.6, permite que el plugin modifique los datos de los ficheros según se envían o reciben, por ejemplo, para implementar cifrado o compresión. Puede utilizarse conjuntamente con el plugin anterior para almacenar información sobre el tratamiento realizado a los datos.

Para evitar cargar todo el fichero en memoria, se utilizan clases derivadas de `QIODevice`, recibiendo el plugin uno correspondiente a el fichero en disco o al canal de red que está descargando el fichero, y debiendo devolver una implementación de una subclase propia del plugin de manera que, al leer sobre el objeto del plugin, se produzca una lectura sobre el objeto pasado por parámetro, efectuando las modificaciones deseadas antes de devolver los datos.

7.5.3. Transferencia de ficheros

La implementación por defecto de la transferencia de ficheros simplemente envía todos los datos de este, sin ninguna clase de compresión. Para acelerar

```
class DataModPlugin : public BasePlugin {
public:
        virtual QIODevice *getUploadDevice(QIODevice *dev
            ) = 0;
        virtual QIODevice *getDownloadDevice(QIODevice *
            dev) = 0;
};

Q_DECLARE_INTERFACE(DataModPlugin, "bsync.client.datamod
    /1.0");
```

Listado de código 7.6: Interfaz para el plugin de modificación de datos

este proceso, la transferencia se puede extender mediante el uso de plugins que implementen la interfaz mostrada en el listado de código 7.7 para implementar algoritmos que reduzcan la cantidad de datos a enviar, por ejemplo, el algoritmo rsync comentado en la subsección 2.2.2.

```
class TransferPlugin : public BasePlugin {
public:
        virtual void *startUpload(QIODevice *file,
            QIODevice *channel) = 0;
        virtual void *startDownload(QIODevice *file,
            QIODevice *channel) = 0;
/* Use the following signals for notifications
signals:
        void uploadFinished(bool ok);
        void downloadFinished(bool ok);
*/
};

Q_DECLARE_INTERFACE(TransferPlugin, "bsync.transfer/1.0")
    ;
```

Listado de código 7.7: Interfaz para el plugin de transferencia de ficheros

Evidentemente, para el buen funcionamiento de la transferencia, es necesario que tanto cliente como servidor utilicen el mismo plugin, puesto que la codificación de envío debe ser poder deshecha por el receptor. Para iniciar una transferencia el programa principal simplemente llama al método correspondiente del plugin que queda a cargo de gestionar la transferencia, y de emitir la señal correspondiente cuando ésta finalice con éxito o se produzca un error.

7.5.4. Comunicación arbitraria

Los comportamientos más complejos pueden obtenerse mediante la utilización de este tipo de plugins cuya interfaz se muestra en el listado de código 7.8, ya que permite la creación de canales de comunicación arbitrarios que pueden

utilizarse para implementar cualquier funcionalidad, como por ejemplo, un sistema de chat o una red privada virtual (VPN).

```
class ChannelInterface {
        virtual QByteArray *openChannel(Share::share_id
            share, QString name, bool data = false) = 0;
}

class ChannelPlugin : public BasePlugin {
public:
        virtual void initChannels(ChannelInterface *iface
            );
};

Q_DECLARE_INTERFACE(ChannelPlugin, "bsync.client.channel
    /1.0");
```

Listado de código 7.8: Interfaz para el plugin de manejo de canales

La interfaz principal tan sólo pasa un puntero a la clase `ChannelInterface`, implementada por el programa principal, de manera que el plugin pueda acceder a la funcionalidad de abrir plugins. Esta clase de interfaz tan sólo proporciona un método para abrir canales, siendo un parámetro el que decide si son canales de datos o de control.

Cada canal lleva asociado un nombre y una compartición, de manera que el servidor retransmitirá automáticamente todos los mensajes recibidos a través de ese canal a todos los clientes que tengan abierto un canal con el mismo nombre y compartición.

Capítulo 8

Interfaz web

Se ha decidido desarrollar una interfaz web para permitir el acceso a los datos del sistema sin necesidad de instalar el cliente de sincronización. En este capítulo se describen las tecnologías utilizadas para la creación de esta interfaz, así como la integración con el servidor y posibles mejoras.

8.1. Plataforma de desarrollo

A pesar de poder usar la misma plataforma que en el resto del sistema, C++ y Qt, para crear la interfaz web haciendo uso de la interfaz fastcgi, se ha decidido utilizar un framework web, mucho más adecuado para la tarea y que reducirá de manera importante el tiempo de desarrollo, al implementar muchas de las funcionalidades comunes.

En primer lugar se desea un lenguaje de tipado dinámico, que suelen ser más adecuados para desarrollo web dado que la velocidad de desarrollo suele ser mayor y en una aplicación de este tipo, que no opera con muchos datos, el tipado dinámico no suele ser la causa de muchos problemas.

Los lenguajes de este tipo más populares para desarrollo web son probablemente Perl, PHP, Python y Ruby. Dado que el objetivo de la interfaz web es interactuar con el sistema, sería conveniente que el lenguaje pudiera ser usado junto a Qt, con el objetivo de facilitar la comunicación web-servidor. Esto no es imprescindible puesto que la mayor parte de la interacción se llevará a cabo a través de consultas a la base de datos, pero futuras extenesiones a la interfaz web pueden requerir, por ejemplo, de acceso a plugins del servidor, lo que se facilita utilizando Qt.

Por eso, se escoge como lenguaje de programación Python, por tener los *bindings* de Qt más desarrollados de los cuatro, siendo Python una de los tres lenguajes preferentes para Qt: C++, Java y Python.

Usar Python sin librerías para facilitar el desarrollo web es mejor que C++, pero el tiempo de desarrollo puede reducirse realmente utilizando una de estas para no tener que recrear partes comunes de casi toda aplicación como el sistema de login o la separación de código de programa y HTML.

El framework elegido es Django [31] al ser uno de los maduros e integrar gran cantidad de componentes especialmente orientados a la creación de aplicaciones de sólo-lectura como blogs o periódicos, puesto que se originó como solución

precisamente para esto. Utiliza el patrón MVC (Modelo-Vista-Controlador), que divide el código de la aplicación en tres partes:

- **Modelo**: Representación de datos, acceso a la base de datos.
- **Vista**: Formateo de datos para el usuario, plantillas HTML.
- **Controlador**: Manipulación de datos y lógica de la aplicación.

Sin embargo, Django usa una nomenclatura distinta para estos tres componentes: el modelo mantiene el nombre, pero las vistas se denominan plantillas y los controladores se pasan a llamar vistas, lo que asegura un grado alto de confusión. Este sistema de nombres intenta reflejar el énfasis en aplicaciones de acceso a datos, en las que la mayor parte del código del controlador no hará más que cargar los datos del modelo y pasarlos a las plantillas para su envío al usuario. A partir de este momento, se utilizará la nomenclatura de Django durante el resto del documento.

8.2. Diseño

Uno de los objetivos principales del diseño de la interfaz web es crear una aplicación moderna, utilizando las técnicas de carga asíncrona (AJAX) de manera que toda la navegación se realice de forma dinámica sin tener que recargar la página a cada click, lo que mejora la experiencia de usuario.

Además, esto evita tener que cargar toda la estructura de directorios de la compartición consultada en una sólo petición del cliente. En cambio, cada vez que se expanda un nodo del árbol de directorios, se cargarán los hijos de ese node, de manera muy similar al árbol de directorios de los exploradores de ficheros. Los detalles de los objetos también pueden ser cargados dinámicamente, evitando tener que entrar en una página de detalles para consultarlos.

Además, dado que no es necesario que el servidor web devuelva directamente HTML, puesto que las funciones de carga dinámica pueden tratar estos datos para mostrarlos adecuadamente, se pueden devolver datos codificados en un formato de intercambio como XML o JSON, de manera que sea más fácil desarrollar herramientas que se aprovechen de esta interfaz para automatizar tareas o desarrollar otras interfaces web para acceder a los mismos datos.

A continuación se pasa a describir el diseño de la interfaz web, describiendo en cada subsección uno de los tres elementos del patrón MVC.

8.2.1. Modelo

La creación del modelo de datos para la interfaz web es trivial, puesto que para poder interoperar correctamente con el sistema ha de ser el mismo que el del servidor ya descrito en la sección 6.2. Simplemente se especifican las tablas ya creadas en el formato requerido por Django por su capa de acceso a datos.

8.2.2. Vista

Las vistas de Django (controladores en MVC estándar) implementan la lógica de la aplicación, que en este caso es muy sencilla al tratase de una aplicación únicamente de consulta, sin modificación de datos.

Autenticación

La primera vista y que permite el acceso a la aplicación es la ocupada de la autenticación del usuario. Para ello se utiliza el sistema de login incluído con Django, que gestiona de manera automática las sesiones y cookies de los usuarios de manera que se conozca el usuario conectado en cada momento.

Sin embargo, ha sido necesario modificar ese sistema para utilizar como backend de autenticación el mismo que en el servidor, puesto que de otro modo habría que mantener dos credenciales para cada usuario: la del cliente de sincronización y la de la interfaz web.

Para facilitar la instalación de la aplicación web, en vez de utilizar el sistema de plugins de Qt para validar el usuario (lo cuál requeriría instalar PyQt, que está disponible en muy pocos servidores), se accede directamente a la tabla utilizada para el almacenamiento de las contraseñas del único plugin de autenticación existente hasta el momento: PgCryptoAuthenticator descrito en la subsección 7.3.2.

Para ello se implementan las funciones requeridas por los backends de autenticación de Django: `login()` que es idéntica a la del plugin de autenticación de bsync y `get_user()` que simplemente devuelve un objeto de la clase `User` para identificar al usuario.

Navegación de los directorios sincronizados

Las vistas principales del sistema son las encargadas de la navegación por los objetos sincronizados del sistema. Dado que la carga de los objetos se realizará de forma asíncrona utilizando AJAX, no es necesario generar todos los resultados de una vez, evitando tener que recorrer todo el árbol de directorios de un cliente cada vez que este realiza una acción.

En cambio, cada petición de navegación sólo devuelve los hijos directos del elemento navegado, es decir, la carga de la raíz devuelve las comparticiones a las que puede acceder un usuario, la apertura de una de estas comparticiones carga la lista de directorios y ficheros en el raíz de esta y la apertura de uno de estos directorios devuelve la lista de los subdirectorios y ficheros contenidos en esta.

Esto simplemente se consigue utilizando el modelo de datos para obtener los elementos hijos de cada elemento, lo que en Django es trivial mediante el uso de los atributos tipo conjunto de cada clase del modelo de datos. Así por ejemplo, la carga de todos los directorios y ficheros contenidos en un share se completa como se muestra en el listado de código 8.1. A este código sólo es necesario añadirle el control de errores, y pasar los datos recuperados a una plantilla para estar completo.

```
share = models.Share(shareid)

directories = share.dir_set.filter(parent=None):
files = share.file_set.filter(parent=None):
```

Listado de código 8.1: Carga de hijos de una compartición en la interfaz web

También funcionan de forma similar las vistas de acceso a los detalles de

los elementos, en las que sencillamente se carga el elemento correspondiente del modelo de datos y se pasa a la plantilla para su representación, siempre con las debidas comprobaciones de que el usuario tiene acceso a dichos datos.

Descarga de fichero

Por último, se permite la descarga de ficheros almacenados en el servidor. Para ello, la vista debe recuperar el identificador de la compartición y el hash del fichero a descargar utilizando el modelo de datos, y a continuación se pasa a ofrecer al cliente la descarga del fichero, que se puede conseguir accediendo al directorio de almacenamiento del servidor.

Se añade el nombre del fichero a las cabeceras HTTP, de manera que el fichero descargado tenga su nombre real y no el de su hash representado en hexadecimal.

8.2.3. Plantillas

La mayoría de las plantillas de este sistema son simplemente contenedores triviales de información en formato JSON, incluyendo los metadatos de los elementos cargados. Se utiliza JSON (JavaScript Object Notation) ya que, como su nombre indica, puede ser interpretado directamente por JavaScript, siendo su tratamiento más eficiente que el de otros formatos como XML.

Sin embargo, además de las plantillas JSON, también son necesarias las interfaces que verá el usuario, que en esta aplicación son únicamente dos. En primer lugar, es necesaria una página para introducir las credenciales, simplemente con un par de campos para el nombre de usuario y al contraseña como se muestra en la figura 8.1.

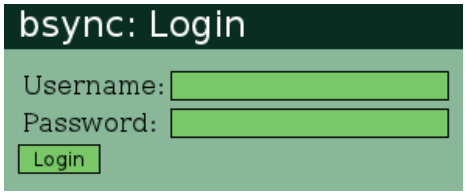


Figura 8.1: Página de login de la interfaz web

En segundo lugar, se encuentra la pantalla principal del sistema, mostrada en la figura 8.2. En la parte superior de esta se encuentra la barra de menús, que en este momento sólo contiene la opción para salir del sistema, lo que devolverá el usuario a la página de login.

El resto de la página se divide en dos mitades: en la parte izquierda se encuentra el árbol de navegación de las comparticiones y en la parte derecha se muestran los detalles del elemento seleccionado.

El árbol de navegación utiliza la librería jsTree [32] que a su vez depende de jQuery [33], y se permite la manipulación dinámica de los elementos representados, incluyendo movimiento, cambios de nombre, etc. Todas las funcionalidades se han desactivado, excepto la carga dinámica de los elementos en formato JSON y los eventos de selección de elementos.

Al pulsar el botón para expandir un elemento, automáticamente se muestra un icono de cargando, mientras que se realiza una petición HTTP a la vista

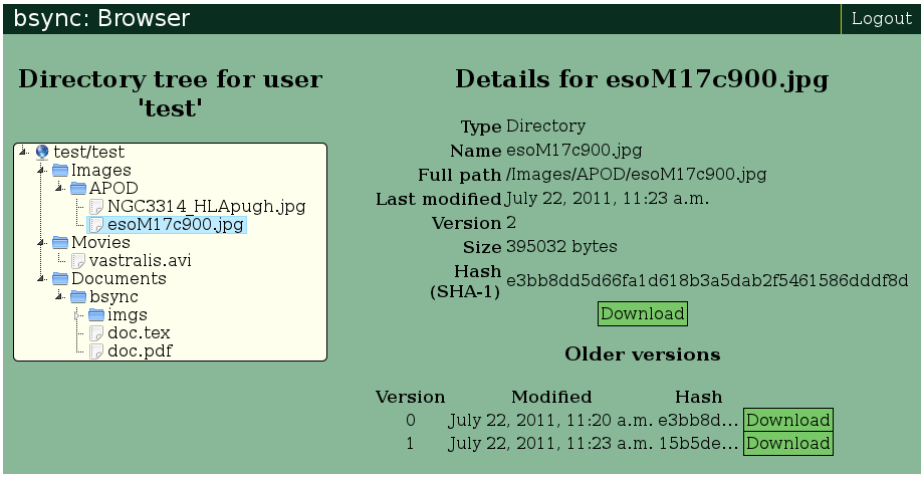


Figura 8.2: Página principal de la interfaz web

ajax correspondiente. Por ejemplo, una petición a `/ajax/share/3` devolverá los ficheros y directorios a nivel de la raíz de la compartición con identificador 3.

De esta forma, se puebla dinámicamente el árbol, purgando automáticamente las ramas usadas hace mayor tiempo en caso de que el número de elementos cargados supere la cantidad máxima de memoria a utilizar por el árbol.

Al seleccionar un elemento, se muestran sus detalles en el panel derecho, incluyendo su tipo, metadatos y, en caso de tratarse de un fichero, un botón para descargarlo. Todos estos datos también se cargan de manera dinámica utilizando AJAX.

También se puede acceder a las versiones antiguas de los objetos, pulsando en un botón dispuesto a tal fin que las cargará dinámicamente mediante una petición HTTP. Estas versiones se muestran en una tabla junto a sus metadatos más relevantes (especialmente la fecha de modifciación) y un botón de descarga en caso de los ficheros.

Las peticiones de descarga tanto de las versiones actuales como antiguas de los ficheros se gestionan con la vista creada a tal fin, y se mostrará el cuadro de descarga estándar del navegador para permitir guardar el fichero de forma local.

Capítulo 9

Seguridad y rendimiento

Este capítulo discute cuestiones generales sobre el rendimiento y la seguridad del sistema. Algunos de los temas tratados ya han sido mencionados en los capítulos anteriores correspondientes al diseño de la parte del sistema correspondiente, aunque en este capítulo se discutirán en profundidad.

Para ello, se localizan los problemas de seguridad y rendimiento más importantes y se proponen varias soluciones, eligiendo siempre la que se considera más adecuada teniendo en cuenta los objetivos del proyecto.

9.1. Hash de los datos

El sistema desarrollado utiliza resúmenes (hashes) criptográficamente seguros para identificar de forma única los datos de los ficheros, al igual que la mayor parte de sistemas similares, como son los de control de versiones o copia de seguridad.

Se utilizan este tipo de funciones por sus características: resumen datos de un dominio indefinidamente grande (entrada de longitud arbitraria) en uno mucho menor y fijo (salida pequeña y de longitud fija), de forma que es altamente improbable que una modificación de los datos no afecte al resumen. También se garantiza una alta complejidad computacional para encontrar mensajes que produzcan un determinado resumen, o encontrar dos mensajes que produzcan el mismo mensaje, es decir, que produzcan una colisión, a pesar de que resulta obvio que existen muchas puesto que el espacio de salida es menor al de entrada.

La resistencia a modificaciones y la improbabilidad de colisiones, hacen estas funcione idóneas para identificar únicamente ficheros, mucho más que códigos de detección y corrección de errores (como los CRC) que sólo garantizan lo primero, siendo por tanto adecuados para comprobar una transferencia exitosa pero no como identificadores únicos.

Existen gran variedad de estas funciones, cada una con características distintas, entre las que destacan para nuestro propósito el nivel de seguridad y la velocidad de procesado de datos que son, en general, medidas opuestas: las funciones más rápidas tienden a ser más inseguras.

Otro parámetro que puede ser modificado relacionado con las colisiones entre datos es el dominio de colisión, es decir, en qué ámbito se considerarán únicos los hashes de los datos. Esto es importante puesto que las posibles colisiones (vo-

luntarias e involuntarias) pueden restringirse a un dominio menor, aumentando la seguridad global del sistema.

9.1.1. Algoritmo de hash

La elección del hash a utilizar para identificar los datos es una decisión importante que puede afectar a la seguridad y al rendimiento e incluso al buen funcionamiento del sistema.

Por un lado, un resumen corto y simple tiende a producir muchas colisiones, lo que ocasiona problemas de funcionamiento (dos ficheros idénticos identificados como iguales, lo que haría que los clientes sólo se bajaran uno de ellos y utilizaran sus datos para los dos ficheros). Además, bajo ciertas circunstancias, esto se puede utilizar en ataques de modificación de datos, ya que las garantías de integridad de los datos son débiles.

Por otro lado, las funciones hash más largas y seguras son también más costosas computacionalmente, lo que ralentizará el sistema, ya que todo fichero tiene que ser hasheado antes de ser enviado al servidor.

La función elegida deberá ofrecer un buen balance entre seguridad y rendimiento, para minimizar las colisiones sin que esto afecte en sobremanera al rendimiento. Para ello, se decide analizar el rendimiento y a la seguridad de varias funciones hash, siendo lo primero relativamente fácil mediante la medición de tiempo. La seguridad es más complicada de medir objetivamente y en general se basa en comparar el mejor ataque disponible contra la función en la actualidad, siendo más seguras aquellas que tienen ataques con mayor complejidad computacional o contra versiones más reducidas de la función.

Presentación de las funciones y análisis de seguridad

Las funciones consideradas son las implementadas en la librería OpenSSL [34], por ser una de las más usadas en el ámbito de la seguridad y provee gran cantidad de funciones. Las funciones, junto con sus características principales se listan en la tabla 9.1.

Algoritmo	**Publicación**	**Salida**	**Rondas**	**Mejor ataque**
md4	1990	128	3	Roto [35, 36]
md5	1992	128	4	Roto [37]
mdc2	1987	128	N/A	2^{124} [38]
ripemd160	1996	160	5	– [39]
sha0	1993	160	80	Roto [40]
sha1	1995	160	80	Roto(2^{51}) [41]
sha224	2001	224	64	–
sha256	2001	256	64	41 rondas [42]
sha384	2001	384	80	–
sha512	2001	512	80	46 rondas [42]
whirlpool	2000	512	10	4,5 rondas [43]

Tabla 9.1: Presentación de algoritmos de hash

De la lista, los algoritmos más conocidos son los de las familias MD y SHA, por lo que han sido más analizados y existen más ataques contra estos, especialmente contra los más antiguos.

De los algoritmos anteriores a 2000, sólo RIPEMD-160 y MDC-2 no están rotos., probablemente debido al poco criptoanálisis recibido al tratarse de un algoritmo patentado. Pero dado que el algoritmo consiste en una variación del cifrado simétrico DES, que puede ser atacado por fuerza bruta en la actualidad, es probable que no sea muy seguro.

Tampoco existen ataques contra RIPEMD-160, aunque sí para sus versiones más cortas: RIPEMD y RIPEMD-128, aunque los ataques contra estos no parecen aplicables a la versión de 160 bits, por lo que de momento se considera seguro.

Para los algoritmos más modernos (posteriores al 2000) existen algunos ataques pero para versiones reducidas de los algoritmos, aproximadamente con la mitad de rondas. A pesar de que SHA-224 es una versión truncada de SHA-256, al igual que lo es SHA-384 de SHA-512, los ataques descubiertos contra las versiones completas no pueden aplicarse a las versiones truncadas. No se han publicado ataques contra estas últimas.

Respecto a su disponibilidad, MD5 y SHA-1 son las más comunes, estando implementadas en la mayoría de librerías e incluso en los sistemas operativos para distintas labores que precisan de integridad. También empiezan a ser comunes las funciones de la familia SHA-2, como sustitutas contras las anteriores que se consideran rotas y no recomendadas para nuevas aplicaciones.

Rendimiento

Para analizar el rendimiento, se ha experimentado utilizando las distintas funciones hash sobre el mismo conjunto de datos, tal y como lo haría el programa cliente. Como cabría esperar, el tiempo empleado por la función es linealmente dependiente del tamaño de los datos a resumir, por lo que se ha considerado que un sólo tamaño de fichero es suficiente para realizar las pruebas.

El tamaño elegido debe ser suficientemente pequeño como para poder contenerse completamente en memoria RAM para poder estudiar los efectos de la caché sobre el tiempo de ejecución de las funciones, de forma que se pueda diferencia el tiempo de entrada y salida del de procesamiento. Además debe ser suficientemente grande como para que la ejecución de las funciones sea suficientemente larga ($> 100ms$) como para que las medidas sean suficientemente precisas. El tamaño elegido es de 150MiB, que arroja los resultados mostrados en la tabla 9.2, siendo la unidad de tiempo el segundo.

Todos los tiempos se miden en segundos y representan la media de 10 ejecuciones de la función en cada categoría (con y sin caché). Para borrar la caché del sistema operativo se escribe 3 al fichero `/proc/sys/vm/drop_caches` en Linux. Los tiempos se desglosan en el total (real), el tiempo en código de usuario (user) y el tiempo empleado en llamadas al sistema (sys), correspondiendo los dos últimos aproximadamente al tiempo de la función resumen y al de entrada/salida respectivamente.

Se observa un aumento de tiempo empleado en la función resumen según se desciende en la tabla, cuyo orden corresponde aproximadamente con la fecha de aparición de los algoritmos, excepto por MDC-2 que obtiene resultados mucho más lentos que cualquier otra función. Los tiempos empleados en llamadas al sistema se mantienen aproximadamente constantes.

También se observa, que MD4, MD5, RIPEMD-160, SHA-0 y SHA-1 emplean aproximadamente el mismo tiempo total, lo cuál se observa claramente en la

Algoritmo	Sin caché			Con caché		
	real	**user**	**sys**	**real**	**user**	**sys**
md4	1,0976	0,2642	0,1294	0,2684	0,1862	0,0814
md5	1,0400	0,3126	0,1252	0,3634	0,2872	0,0742
mdc2	17,5050	17,1912	0,1878	17,2996	17,1630	0,1290
ripemd160	1,0794	0,8148	0,1548	0,8836	0,8112	0,0700
sha0	1,0334	0,6820	0,1114	0,7106	0,6420	0,0658
sha1	1,0212	0,4566	0,1228	0,5020	0,4216	0,0786
sha224	1,3718	1,0874	0,1754	1,1748	1,1000	0,0726
sha256	1,3742	1,1014	0,1606	1,1726	1,0946	0,0748
sha384	1,4854	1,2192	0,1588	1,2952	1,2246	0,0680
sha512	1,5284	1,2126	0,1626	1,2914	1,2146	0,0732
whirlpool	2,2882	2,0012	0,1640	2,0760	1,9998	0,0726

Tabla 9.2: Comparación de rendimiento de algoritmos de hash

figura 9.1. Dado que este fenómeno sólo se observa en los tiempos que no utilizan la caché del fichero a resumir, puede deducirse que este tiempo (algo superior a un segundo) es el que tarda el sistema de pruebas en leer el fichero de disco, siendo este el factor limitante en el proceso.

9.1.2. Ataques y dominio de colisión

El dominio de colisión es aquel en el que dos conjuntos de datos con mismo resumen se consideran idénticos. Dentro del mismo dominio, los objetos con los mismos datos (según el resumen) se almacenan una única vez, lo que tiene dos consecuencia principales:

- Dos ficheros con distintos datos y mismo resumen colisionarán, y sólo se almacenarán los datos de uno.
- Dos ficheros con exactamente los mismos datos se almacenarán una sola vez.

Ambas consecuencias son dos vistas al mismo hecho: los datos se identifican por su hash. Mientras que la primera consecuencia debería evitarse, la segunda es favorable, puesto que ahorra espacio en el dispositivo de almacenamiento del servidor.

Las colisiones aleatorias son poco probables por las características de diseño de los algoritmos de hash, sin embargo, algunos de estos algoritmos son vulnerables a ataques que permiten encontrar colisiones para ficheros dados. En caso de que el dominio fuera muy amplio (por ejemplo: todo el servidor), se podrían usar estos ataques para crear ficheros maliciosos con el mismo resumen que ficheros populares (películas, canciones...), lo que provocaría la descarga del fichero malicioso en escenarios como el siguiente:

1. Dado un fichero popular P, su resumen H y un fichero malicioso M con el mismo hash.
2. El atacante sincroniza un directorio que contenga M.

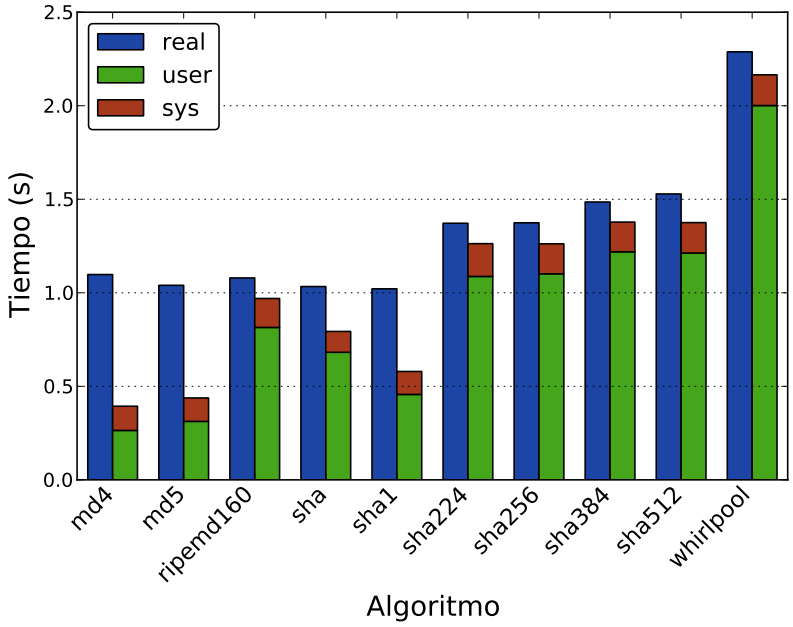


Figura 9.1: Comparación de rendimiento de algoritmos de hash (sin caché)

3. Un usuario añade el fichero P a su directorio de sincronización de un equipo. El cliente no tendrá que subir el fichero al servidor, ya que este le comunicará que ya tiene un fichero para el hash H.

4. Otro equipo del usuario recibe la notificación de nuevo fichero creado con hash H y procede a descargarse el fichero correspondiente, que en este caso es M.

5. El fichero malicioso aparece en el directorio sincronizado de uno de los dos equipos del cliente.

Este ataque se puede evitar si se reduce el dominio de colisión a nivel de compartición. De esta manera, el atacante tendría un fichero M con hash H en el share S_m, mientras que el usuario legítimo podría sincronizar el fichero P con hash H en el share S_p.

Sin embargo, esto tiene el inconveniente que en el caso de que varios usuarios sincronicen el fichero P, se almacenará una copia por cada usuario en el servidor, mientras que en el caso del dominio a nivel de servidor, todos los usuarios compartirían una sola copia.

9.1.3. Decisión

La decisión sobre el dominio de colisión y el algoritmo de hash a utilizar se hace en conjunto, puesto que un dominio más restrictivo permite utilizar un hash menos seguro, ya que el daño que un atacante puede realizar mediante la búsqueda de colisiones está limitado al dominio de colisión.

Un dominio restringido permite utilizar algoritmos más rápidos (menor gasto computacional en el cliente) pero a cambio se aumenta la capacidad de almacenamiento necesaria en el servidor. En caso de utilizarse un dominio más amplio es necesario utilizar un algoritmo de hash más seguro y por tanto más lento, lo que aumenta el uso de la CPU en el servidor a cambio de reducir el espacio de almacenamiento del servidor.

Por otro lado, los algoritmos de hash y los ataques contra estos están en constante evolución y un ataque contra un algoritmo considerado seguro puede ser descubierto en cualquier momento, con lo que es necesario actualizar el algoritmo utilizado por el sistema periódicamente.

Sin embargo, la seguridad proporcionada por un dominio más reducido no se ve afectada por el paso del tiempo, con lo que es mejor opción de cara el futuro. Por esto, se elige utilizar un dominio reducido (a nivel de compartición) con un algoritmo más ligero. Esto evitará colisiones con otros usuarios del sistema, y se considera que en los escenarios de aplicación más probables para este sistema (empresas y otras instituciones) no es probable que existan muchos ficheros localizados en múltiples comparticiones (cada fichero debería encontrarse en la compartición correspondiente al grupo de trabajo responsable), mitigando el uso incrementado de espacio en el servidor.

Para los algoritmos de hash se elige una estrategia que permitirá utilizar nuevos algoritmos según sea necesario, mediante la introducción de un byte adicional al comienzo del campo resumen indicando el algoritmo utilizado. De entre los algoritmos actuales, se elige SHA-1 por ser uno de los más rápidos, con la mejor seguridad de la categoría de algoritmos con menor consumo de tiempo en CPU que en entrada y salida, a excepción de RIPEMD-160 que no ha sido tan escrutado criptográficamente.

Además, es uno de los algoritmos más extendidos, estando implementado en multitud de librerías y sistemas operativos, lo que lo hace una buena opción como algoritmo mínimo que deba ser implementado por todos los clientes que deseen acceder al sistema. De hecho, es el único junto con MD5 y MD4 que está integrado en las librerías de Qt sin necesidad de añadir la dependencia de OpenSSL, lo cuál facilita la distribución del código a países con restricciones de uso o patentes sobre los algoritmos implementados en dicha librería.

9.2. Almacenamiento de contraseñas

El sistema utiliza y almacena contraseñas en ambos componentes: cliente y servidor. El cliente necesita almacenar la contraseña para autenticarse sin necesidad de que el usuario la teclee cada vez, mientras que el servidor la guarda para poder verificar las credenciales del usuario.

9.2.1. Cliente

En el lado del cliente, es imprescindible tener acceso a la contraseña original, por lo que si no se desea ninguna intervención por parte del usuario no es posible cifrar la contraseña de ninguna forma, siendo sólo posible su ofuscación, por ejemplo: cifrando con una contraseña prefijada o almacenada en algún lugar.

Probablemente la mejor opción sería la utilización de una contraseña maestra para cifrar todas las contraseñas de los múltiples servidores a los que puede

desear conectarse el usuario, ya sea esta fija o pedida al usuario en cada ejecución. La contraseña fija tiene poco sentido, puesto que no consigue más que ofuscar los datos, fácilmente recuperables analizando el binario del programa, y aún más trivialmente si se dispone del código fuente.

La petición al usuario es un buen balance entre seguridad y comodidad (sólo se pide una contraseña al usuario, en vez de una por servidor), sin embargo, no se ha implementado en el prototipo para facilitar la realización de pruebas sin tener que introducir la contraseña cada vez. En caso de comercializar o distribuir el producto, este sería uno de los puntos a mejorar.

9.2.2. Servidor

El almacenamiento de contraseñas en el servidor permite el uso de técnicas criptográficas avanzadas, puesto que no se requiere poder recuperar las contraseñas, sólo poder compararlas con las enviadas por el usuario.

El plugin de autenticación incluido con el prototipo, utiliza las técnicas de criptografía adecuadas para asegurarse la confidencialidad de la contraseña, es decir, la contraseña se guarda utilizando una función hash criptográficamente segura, añadiendo antes de aplicar la función una cadena aleatorio o *salt* para evitar ataques por diccionario o *rainbow tables*.

Para esta aplicación, los criterios de selección son totalmente distintos a los necesarios para el resumen de datos de ficheros, lo que hace que el análisis de la sección 9.1 no sea aplicable. En este caso se desea una función lenta para dificultar los ataques por fuerza bruta, mientras que en el otro caso el objetivo era un buen rendimiento para acelerar la inicialización del punto de montaje.

Entre las alternativas consideradas se encuentran las funciones más lentas analizadas anteriormente (como mdc2), o su utilización sucesiva en múltiples rondas. La función `crypt()` de Unix, en la implementación de glibc soporta varios de estos esquemas, destacando bcrypt [29] (basado en Blowfish) y los basados en SHA-2 [30]. Ambos utilizan múltiples rondas y añaden *salt* a los datos, siendo el número de rondas configurable y el tamaño de salt equivalente al tamaño de la clave y la salida respectivamente.

Dado la flexibilidad de esta alternativa, su alta disponibilidad al estar implementada en glibc, el hecho de poder set utilizada directamente desde varios motores de base de datos y el gran grado de seguridad que proporciona al aplicar las técnicas criptográficas adecuadas, hacen que sea la utilizada para el plugin de autenticación suministrado con el prototipo.

9.3. Seguridad de las comunicaciones

La seguridad en las comunicaciones se garantiza mediante el empleo del protocolo de transporte TLS, sucesor de SSL, por lo que la seguridad de las comunicaciones del sistema tendrá las características ofrecidas por este.

TLS provee servicios de confidencialidad, integridad y autenticación que garantizan que los datos enviados no pueden ser leídos por terceras partes, que estos son correctos y que el servidor y el cliente son los que dicen ser respectivamente. Lo primero es necesario para evitar filtrar los datos contenidos en los ficheros a terceras partes, lo segundo para garantizar que los envíos de ficheros y notificaciones son correctos y lo tercero para evitar que los clientes tomen como

auténtico un servidor que no lo es, evitando ataques Man-In-The-Middle entre otros.

A pesar de que TLS puede utilizarse para autenticación mutua, el sistema sólo lo utiliza para autenticar el servidor frente al cliente, mediante la utilización de un certificado de clavo pública. El cliente también podría autenticarse utilizando el mismo sistema, pero la implementación del prototipo utiliza credenciales en forma de usuario y contraseña para esto, puesto que se entiende que son más compatibles con diversos sistemas de autenticación.

En las transferencias de ficheros se implementa además un control adicional de integridad utilizando el hash de los datos del fichero, lo cuál aumenta la seguridad de una transferencia completa a nivel de fichero (en vez de a nivel de bloque como hace TLS), garantizando además que no se ha producido una modificación adicional en el fichero temporal.

Dado que tanto el cliente como el servidor utilizan la misma implementación del protocolo, ambos tienen disponibles los mismos algoritmos de seguridad, por lo que en la negociación se selecciona automáticamente el que se considera más seguro en la implementación, en este caso `AES256-SHA`, es decir, cifrado por bloques utilizando AES de 256 bits en modo CBC y control de integridad utilizando SHA-1.

9.4. Paralelismo

Paara asegurar un buen rendimiento del sistema en sistemas modernos, es imprescindible la buena utilización de múltiples hilos de ejecución ya que la tendencia actual es a aumentar el número de unidades de procesamiento en la CPU, en vez de su frecuencia de funcionamiento.

Tanto el cliente como el servidor utilizan esta técnica, aunque de forma más agresiva en el servidor que en el cliente. Esto se debe en primer lugar a que el servidor gestionará más conexiones simultáneas, además de disponer, por lo general, de más recursos que los equipos cliente. Además, el cliente no debe suponer una carga importante de trabajo en el cliente, siendo preferible que las tareas tarden más tiempo a que bloqueen el sistema para que la sincronzación de ficheros no afecte al funcionamiento general del sistema.

9.4.1. Cliente

En el cliente, se utilizan hilos con dos finalidades: evitar el bloqueo de la sincronización de unos directorios por otros y acelerar la tarea más lenta del sistema: el cálculo de hash de ficheros. La separación en hilos de las distintas clases del sistema se representa en la figura 9.2.

La implementación de lo segundo es muy sencilla utilizando un conjunto o *pool* de hilos para calcular los resúmenes de varios ficheros simultáneamente. Cada petición de hash va a parar a una cola de tareas, y será ejecutada tan pronto como uno de los hilos del conjunto quede libre. Cuando la tarea se complete, se notificará a la compartición que realizó la petición para que pueda seguir procesando el fichero.

Todo esto se consigue utilizando la clase `QThreadPool` de Qt, que gestiona de forma automática el conjunto de hilos y la cola de tareas (representadas por un `QRunnable`). El número de hilos del conjunto es equivalente por defecto

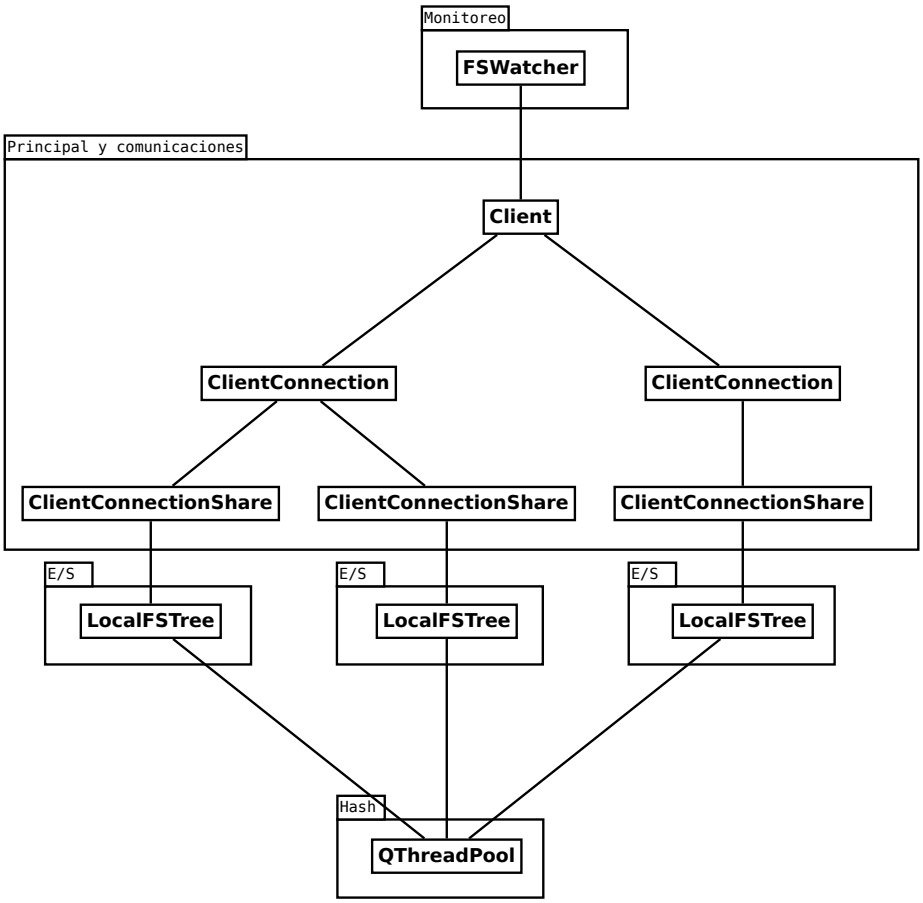


Figura 9.2: Utilización de hilos en el cliente

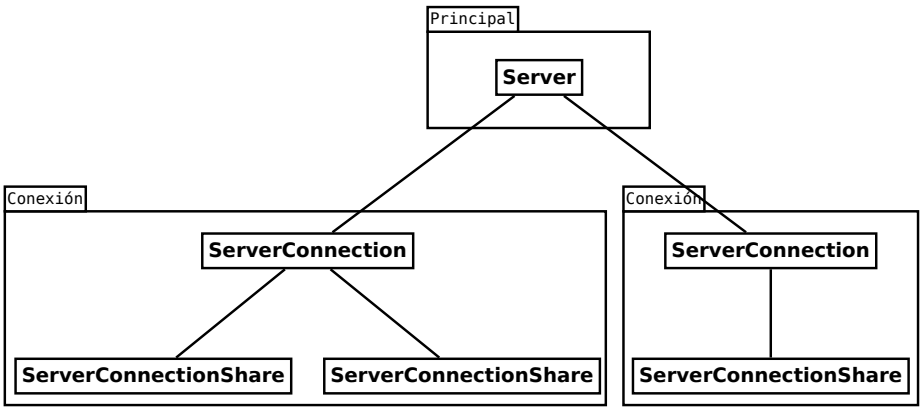


Figura 9.3: Utilización de hilos en el servidor

al número de procesadores o núcleos del equipo, aunque puede ajustarse en el fichero de configuración.

La otra función de los hilos en el cliente es evitar que operaciones lentas en uno de los directorios a sincronizar bloqueen al resto de estos. Esto es un problema si las comunicaciones se ejecutan de forma síncrona con las operaciones de acceso a disco, puesto que la recepción de mensajes de red quedaría bloqueada mientras se ejecutara la operación de acceso a disco.

Para evitar esto, cada instancia de la clase `LocalFSTree` que engloba todas las operaciones sobre el sistema de ficheros se ejecuta en su propio hilo, diferente al de las comunicaciones, de forma que mientras cada compartición ejecuta sus operaciones sobre disco, sea posible seguir recibiendo y enviando notificaciones relativas a otros shares.

La comunicación entre el hilo de comunicaciones (que no es necesario dividir en varios puesto que se ha comprobado que el tiempo de procesado de peticiones es despreciable frente al tiempo de transmisión) y los hilos de entrada/salida se realiza utilizando el ya descrito sistema de señales y slots de Qt, que en caso de conexiones entre distintos hilos utilizará automáticamente una cola de mensajes para cada receptor, de forma que en el momento en el que el hilo pase a estar inactivo se procesará la siguiente petición en la cola. También se utilizan mutexes en los casos en los que no es posible utilizar el mecanismo de señales y slots porque se requiera de respuesta inmediata.

9.4.2. Servidor

La utilización de hilos en el servidor se debe a la necesidad de ser capaz de procesar peticiones de muchos clientes al mismo tiempo, por lo que es adecuado separar en un hilo cada conexión al servidor. Esto es similar al comportamiento de muchos servidores web, como por ejemplo Apache con el módulo mpm_worker [44].

Para ello simplemente se separa cada instancia de la clase `ServerConnection` en un hilo, de forma que todos los métodos de esta clase se ejecutan en dicho hilo, como se muestra en la figura 9.3. El acceso a otro hilos se gestiona como en el cliente, señales y slots cuando es posible y mutexes en otro caso.

El problema de utilizar un hilo por cada conexión es que gestionar miles de conexiones ocasionaría la creación de miles de hilos, con el consiguiente problema de gestión de estos. Para evitar esto, la clase principal del servidor mantiene una lista de hilos abiertos, y en caso de superarse el número máximo de estos especificado en la configuración, se pasará a asignar las nuevas conexiones a hilos que ya tienen asignada otra conexión, de manera que en algunos hilos se gestionen varias conexiones.

Parte IV

Conclusiones y Presupuesto

Capítulo 10

Conclusiones

En este documento se ha descrito un sistema de sincronización y copia de seguridad de ficheros dirigido a las organizaciones y altamente extensible. A pesar de que existen alternativas similares en el estado del arte, la creada durante este proyecto permite la utilización de sistemas de este tipo en un campo más amplio que los existentes, dado que se presenta como un producto completo, incluyendo el servidor de almacenamiento y los clientes de sincronización.

Además, las capacidades de extensión del sistema hacen que pueda utilizarse en múltiples escenarios de sincronización de datos, no necesariamente de ficheros, facilitando la integración con tecnologías ya desplegadas y la personalización a las necesidades propias del usuario final.

El resultado final contiene todas las funcionalidades básicas de los sistemas del estado del arte, ya sea en el núcleo del sistema o mediante plugins: sincronización en tiempo real de directorios entre varios equipos, auto-archivo de versiones antiguas y posibilidad de acceso a los datos a través de una interfaz web además del cliente de escritorio facilitando la portabilidad.

También pueden implementarse características avanzadas utilizando el sistema de plugins, lo que permitirá que estas puedan ser utilizadas sólo cuando se desee. Algunas posibilidades son cifrado y compresión de los datos, adición de metadatos a los objetos sincronizados o utilización de la red existente entre clientes para implementar funcionalidades arbitrarias como puede ser comunicación en tiempo real.

10.1. Trabajo futuro

A pesar de presentarse un sistema completo, existen gran cantidad de puntos que pueden mejorar este sistema. Un punto claro es el desarrollo de mayor número de plugins, tanto a nivel de interfaz como implementaciones utilizando las interfaces ya disponibles. Además, las siguientes ideas concretas son algunas de las más interesantes.

10.1.1. Acceso a comparticiones en otros servidores

Para acceder a comparticiones en un servidor es necesario disponer de credenciales en ese servidor, pertenezca o no la compartición al usuario que desea

acceder a ella. Esto provoca que usuarios de distintos servidores que deseen compartir sus datos precisen de credenciales en todos los servidores en los que haya comparticiones a las que quieran tener acceso.

Para solventar esto se proponen dos alternativas: un sistema completamente federado, al estilo de XMPP o la utilización de una plataforma de login distribuida como OpenID [45].

En la primera solución, cada usuario se conecta a su servidor, y son estos los que establecen redes entre sí para el intercambio de datos, de forma que cada cliente intercambiaría notificaciones y datos únicamente con su servidor, ocupándose este de hacerlo llegar al servidor y clientes correspondientes mediantes conexiones servidor a servidor.

Esto es idéntico al funcionamiento de XMPP, y podrían utilizarse las mismas técnicas de autenticación utilizadas en este como se describen en [46]. El problema de esta solución es el incremento de tráfico en los servidores que no alojan las comparticiones, actuando como proxy entre el usuario y el servidor final, pero facilita el establecimiento de relaciones de confianza de los clientes, al tener que confiar únicamente en su servidor de manera directa.

La otra solución propuesta es mucho más ligera, federándose únicamente la autenticación. En este caso, el cliente se conectaría directamente al servidor en el que no tiene credenciales y utilizaría sus credenciales de otro servidor. Los servidores se comunicarían directamente entre sí para verificar las credenciales del usuario y darle acceso en otro servidor.

Esta solución es más ligera al ser el único tráfico entre servidores el necesario para la autenticación y puede implementarse mediante el uso de plugins de autenticación sin necesidad de modificar el núcleo. El plugin recibiría las credenciales del usuario y se ocuparía de conectarse a su servidor original para validarle. En caso de desear seguir un estándar como OpenID, es probable que fuera necesaria una modificación del núcleo, en caso de que el sistema actual basado en identificador y contraseña no fuera necesario para transmitir la información necesaria para un login exitoso.

10.1.2. Seguridad en los plugins

Ya se ha discutido el grave problema de seguridad que suponen los plugins al poder ejecutar código arbitrario, y como se consigue mitigar el riesgo utilizando plugins únicamente de fuentes de confianza. Sin embargo, no se dispone de ningún medio técnico para verificar esto.

Por ello, se propone la utilización de firmas digitales basadas en criptografía de clave pública para firmar los plugins, de la misma manera que se hace con el software o los drivers en multitud de sistemas operativos. Antes de cargar un plugin, se verificaría su firma (si la tuviera) para permitir al usuario decidir si confía en la autoridad firmante o no.

Esto proporciona los medios técnicos para verificar la procedencia de los plugins, pero estos siguen pudiendo ejecutar código arbitrario. Este segundo hecho puede ser mitigado modificando el sistema de plugins para que se ejecuten en procesos distintos del núcleo del programa, facilitando así el uso de técnicas de *sandbox* para restringir los permisos de ese proceso. Este técnica ya es usada por navegadores modernos con el mismo propósito y gran éxito, teniendo al ventaja adicional de que errores en los plugins no provocan el cierre del programa principal.

10.1.3. Publicación y comercialización

La idea original del proyecto era la publicación del proyecto y la documentación asociada como software libre, para que la comunidad pueda utilizarlo y expandirlo de forma totalmente libre. Además, todas los sistemas de este tipo libres se construyen a partir de varios componentes, con las limitaciones y complicaciones que esto conlleva.

Sin embargo, durante el desarrollo del proyecto, una pequeña empresa se interesó por el producto, y está en estos momentos realizando estudios de viabilidad para la comercialización del producto, especialmente orientado a vender el software a pequeñas empresas y desarrollar plugins personalizados para el mismo en caso de que deseen una funcionalidad adicional no disponible.

En caso de que la idea no salga adelante con la empresa, el código se publicará bajo una licencia estilo BSD, que proporciona la máxima libertad de actuación sobre el código, en alguna plataforma de proyectos como SourceForge o github.

10.2. Conclusiones personales

Este proyecto ha sido una gran labor de integración de distintas tecnologías, al tratarse de un sistema completo que incluye comunicaciones de red, operaciones sobre ficheros, utilización de bases de datos, nociones de seguridad y paralelismo y diseño de interfaces de usuario, tanto de escritorio como web.

De hecho, cubre prácticamente la totalidad de las materias impartidas en la titulación, exceptuando las de inteligencia artificial. Esto ha permitido completar mi formación, especialmente en estas materias que se alejaban más de mi especialización en inteligencia artificial, descubriendo la utilización práctica de muchos conocimientos adquiridos durante los cinco años de carrera y completando las prácticas de estas asignaturas.

Además de consolidar estos conocimientos, he adquirido otros nuevos, especialmente en lo relativo a la carga dinámica de plugins y los problemas del desarrollo multiplataforma, especialmente evidentes en la necesidad de crear diversos plugins para la misma función: el monitoreo del sistema de ficheros.

Creo que todo esto me ha hecho mejorar como ingeniero, sobre todo como poseedor de conocimientos sobre gran cantidad de disciplinas dentro de la informática lo que, sin duda, ayudará a mejorar los proyectos futuros, mediante el conocimiento de la tecnología idónea a utilizar para resolver cada problema.

Queda aún mucho por aprender y la base de conocimiento cada vez es mayor; la carrera está perdida. Pero lo importante no es llegar a la meta, si no todo lo aprendido durante el camino.

Capítulo 11

Presupuesto

En este capítulo se calcula el presupuesto del proyecto, incluyendo su desglose en distintos tipos de gastos, tanto económicos como de tiempo. Todos los cálculos se efectúan con sus valores antes de impuestos y se expresan en euros.

11.1. Gastos directos

En primer lugar se consideran los gastos directos, es decir, aquellos que se pueden imputar directamente al proyecto. Esta categoría incluye los gastos de personal, la amortización de equipos y el material fungible utilizado por el proyecto.

11.1.1. Gastos de personal

Al tratarse de un proyecto informático, el gasto directo principal es el coste del personal. Para calcular este, se estima el tiempo ocupado por cada fase del proyecto, representado en el diagrama de Gannt de la figura 11.1. Estos tiempos están calculados para una sola persona con una dedicación parcial de alrededor de 20 horas semanales, o el 50 % de una jornada completa.

Se ha tomado el salario medio correspondiente a un analista-programador según Infojobs Trends [47], que es de unos 29.000€ brutos anuales. El coste final se calcula multiplicando el coste anual por la duración del proyecto (18 semanas) y la dedicación del personal al proyecto (50 %):

$$C_{personal} = 29000€ \frac{18}{52} 0{,}5 = 5019,23€$$

11.1.2. Otros gastos directos

También es necesario computar otros gastos directos además de los de personal, en este caso la amortización de los equipos informáticos utilizados. Dado que se todo el proyecto se ha desarrollado con software libre (Linux, QtCreator, gcc, gdb, dia, kile y latex principalmente), no es necesario imputar gastos debidos a licencias de software, siendo únicamente necesario computar el gasto del hardware.

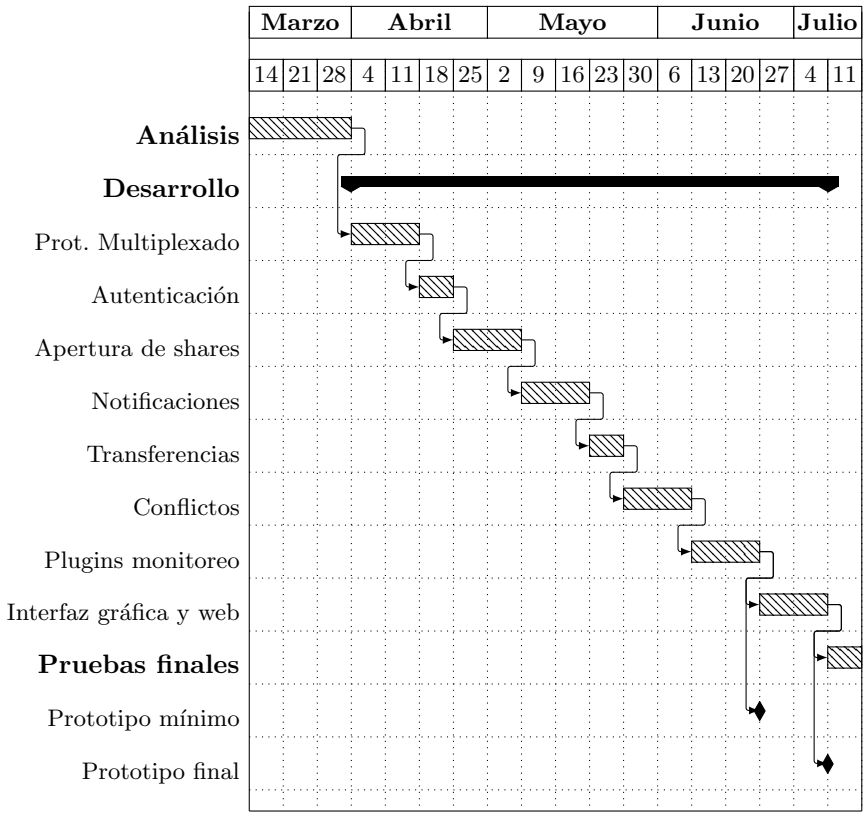


Figura 11.1: Diagrama de Gantt del proyecto

El desarrollo se lleva a cabo en un ordenador portátil ASUS X71SL, adquirido por 699€ en Octubre de 2009 lo que, restando el 16 % de IVA vigente en dicha fecha, equivale a 602,59€ antes de impuestos. El periodo de amortización del equipos es de 3 años por lo que el coste imputable al proyecto, teniendo en cuenta que se utiliza en varios proyectos con dedicación igual a la del personal, será de:

$$C_{hardware} = 602,59€ \frac{18}{52 * 3} 0{,}5 = 34,76€$$

Se considera también el material fungible, en este caso mínimo pues se han utilizado políticas de conservación del medio ambiente, evitando imprimir los documentos y consultándolos en formato electrónico. El único concepto en esta categoría es la compra de un lote de material de papelería por 12,72€.

11.2. Gastos indirectos, riesgo y beneficio

Se añade también al presupuesto los costes indirectos asociados al proyecto, lo que incluye conceptos que son difíciles de imputar directamente al proyecto, tales como el mantenimiento del local o el coste de la electricidad. Para ello se utiliza un porcentaje del 20 % sobre los gastos directos.

Finalmente, se añade un porcentaje de riesgo, con objeto de sufragar posibles errores en el presupuesto. Dada la corta duración del proyecto, un retraso de una semana representa aproximadamente un 5 % de error. Para tener un margen de error de al menos dos semanas, se utilizará como riesgo el 10 % de los costes totales (directos e indirectos) del proyecto.

No hay que olvidar que el objetivo del proyecto es ganar dinero, por lo que se suma al coste total del proyecto el margen de beneficio a obtener, que en este caso se establece al 10 % de los gastos del proyecto.

11.3. Tabla resumen y totales

En la tabla 11.1 se resumen y totalizan los gastos del proyecto detallados en las secciones anteriores.

Concepto	**Base**	**Porcentaje**	**Total**
Total gastos directos			**5066,71**
Gastos de personal			5019,23
Amortización de equipos			34,76
Material fungible			12,72
Gastos indirectos	5066,71	20 %	1013,34
Total gastos			**6080,05**
Riesgo	6080,05	10 %	608,01
Beneficio	6080,05	10 %	608,01
Total			**7296,07**

Tabla 11.1: Tabla resumen del presupuesto

El coste total del proyecto es de siete mil doscientos noventa y seis euros y siete céntimos (7296,07€) sin incluir el impuesto sobre el valor añadido. Aplicando el IVA vigente a Julio de 2011 (18 %) el total es de ocho mil seiscientos nueve euros y treinta y seis céntimos (8609,36€), aunque siempre se aplicará el IVA vigente en el momento del pago del proyecto.

Glosario

- **AJAX** Siglas en inglés de JavaScript asíncrono y XML, se refiere a la utilización de peticiones HTTP de forma asíncrona siendo o no el formato de respuesta XML.
- **API** Siglas en inglés de interfaz de programación de aplicaciones, conjunto de funcionalidad de un sistema expuesto al programador.
- **Cliente** Programa ejecutado en los equipos a sincronizar.
- **Compartición** Directorio sincronizado en el servidor (ver sección 3.4).
- **Hash** Secuencia de bits que identifica de forma única un conjunto de datos (ver sección 9.1).
- **Plugin** Librería dinámica que extiende el funcionamiento del programa.
- **Punto de montaje** Directorio local en el que se sincroniza una compartición (ver sección 3.4).
- **Resumen** Hash.
- **Señal** Utilizada para la señalización de eventos, puede ser conectada a un slot (ver sección 3.8.3).
- **Servidor** Programa ocupado de almacenar los datos sincronizados.
- **Share** Compartición.
- **Slot** Método que puede ser conectado a señales (ver sección 3.8.3).

Bibliografía

[1] A. Chervenak, V. Vellanki, Z. Kurmas; Protecting file systems: A survey of backup techniques. Joint NASA and IEEE Mass Storage Conference (Vol. 3, pp. 17-32), 1998.

[2] Z. Kurmas, A.L. Chervenak; Evaluating Backup Algorithms. 2000.

[3] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, Zambeel Inc., D. Noveck; Nfs Version 4 Protocol. 2000.

[4] Introducción a SMB (Server Message Block)
`http://msdn.microsoft.com/en-us/library/aa365233(VS.85).aspx`

[5] A. Tridgell, P. Mackerras; The rsync algorithm. Australian National University, 1996.

[6] Dropbox
`http://www.dropbox.com/`

[7] Ubuntu One
`https://one.ubuntu.com/`

[8] Tarsnap
`http://www.tarsnap.com/`

[9] lsyncd (Live Syncing Daemon)
`http://code.google.com/p/lsyncd/`

[10] Jungle Disk
`https://www.jungledisk.com/`

[11] SparkleShare
`http://sparkleshare.org/`

[12] TLS (Transport Layer Security); RFC5246
`http://tools.ietf.org/html/rfc5246`

[13] MTLS (Multiplexed Transport Layer Security)
`http://tools.ietf.org/html/draft-badra-hajjeh-mtls-05`

[14] Java, Oracle
`http://www.java.com/`

[15] Qt, Nokia
`http://qt.nokia.com/`

[16] .Net, Microsoft
http://www.microsoft.com/NET/

[17] Mono
http://www.mono-project.com/

[18] wxWidgets
http://www.wxwidgets.org/

[19] Formato de serialización de Java
http://download.oracle.com/javase/6/docs/platform/serialization/spec/protocol.html

[20] Formato de serialización de Qt
http://doc.qt.nokia.com/4.7/datastreamformat.html

[21] Documentación sobre señales y slots de Qt
http://doc.qt.nokia.com/4.7/signalsandslots.html

[22] RFC 5905: Network Time Protocol Version 4: Protocol and Algorithms Specification, IETF.

http://www.ietf.org/rfc/rfc5905.txt
Veansé también RFCs 5906, 5907 y 5908 de la misma fuente.

[23] Motor de base de datos SQLITE
http://www.sqlite.org/

[24] Monitor del sistema de ficheros de Qt
http://doc.trolltech.com/4.7/qfilesystemwatcher.html

[25] Documentación de inotify (inotify.txt)
http://www.kernel.org/doc/Documentation/filesystems/inotify.txt

[26] Documentación de ReadDirectoryChangesW
http://msdn.microsoft.com/en-us/library/aa365465(v=VS.85).aspx

[27] JNotify para Windows
http://jnotify.sourceforge.net/windows.html

[28] Documentación de FSEvents
http://developer.apple.com/library/mac/#documentation/Darwin/Conceptual/FSEvents_ProgGuide/Introduction/Introduction.html

[29] N. Provos; D. Mazières; A Future-Adaptable Password Scheme. Proceedings of USENIX Annual Technical Conference, 1999.

[30] Algoritmo de `crypt()` para SHA-2
http://www.akkadia.org/drepper/SHA-crypt.txt

[31] Django
https://www.djangoproject.com/

[32] jsTree
http://www.jstree.com/

[33] jQuery
http://jquery.com/

[34] OpenSSL
http://www.openssl.org/

[35] Y. Sasaki, L. Wang, K. Ohta and N. Kunihiro; New Message Difference for MD4. Fast Software Encryption (Vol. 4539, pp. 329-348), 2008.

[36] G. Leurent; MD4 is Not One-Way. Fast Software Encryption (Vol. 5086, pp. 412-428), 2008.

[37] S. Chen, C. Jin; An Improved Collision Attack on MD5 Algorithm. Information Security and Cryptology (Vol. 4990, pp. 343-357), 2008.

[38] L. Knudsen, F. Mendel, C. Rechberger, S. Thomsen; Cryptanalysis of MDC-2. Advances in Cryptology - EUROCRYPT 2009 (Vol. 5479, pp. 106-120), 2009.

[39] F. Mendel, N. Pramstaller, C. Rechberger, V. Rijmen; On the Collision Resistance of RIPEMD-160. Information Security (Vol. 4176, pp. 101-116), 2006.

[40] S. Manuel, T. Peyrin; Collisions on SHA-0 in One Hour. Fast Software Encryption (Vol. 5086, pp. 16-35), 2008.

[41] S. Manuel; Classification and generation of disturbance vectors for collision attacks against SHA-1. Designs, Codes and Cryptography (Vol. 59, pp. 247-263), 2011.

[42] Y. Sasak, L. Wang, K. Aoki; Preimage Attacks on 41-Step SHA-256 and 46-Step SHA-512. 2009.

[43] F. Mendel1, C. Rechberger, M. Schläffer, S. Thomsen; Cryptanalysis of Reduced Whirlpool and Grøstl. Fast Software Encryption: 16th International Workshop, 2009.

[44] Módulo mpm_worker de Apache.
http://httpd.apache.org/docs/2.0/mod/worker.html

[45] D. Recordon, D. Reed; OpenID 2.0: a platform for user-centric identity management. Proceedings of the second ACM workshop on Digital identity management (pp. 11-16). 2006.

[46] XEP-0220: Server Dialback
http://xmpp.org/extensions/xep-0220.html

[47] Infojobs trends
http://salarios.infojobs.net/